Software Engineering Institute

Java Concurrency Guidelines

Fred Long
Dhruv Mohindra
Robert Seacord
David Svoboda

May 2010

TECHNICAL REPORT
CMU/SEI-2010-TR-015
ESC-TR-2010-015

CERT® Program

http://www.cert.org/

Carnegie Mellon

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

Table of Contents

Acknowledgments xi

About This Report xiii

Abstract xv

1 Introduction 1

1.1.1 The `volatile` Keyword 4
1.1.2 Synchronization 5
1.1.3 The `java.util.concurrent` Classes 6

2 Visibility and Atomicity (VNA) Guidelines 9

2.1 VNA00-J. Ensure visibility when accessing shared primitive variables 9
2.1.1 Noncompliant Code Example (Non-Volatile Flag) 9
2.1.2 Compliant Solution (`volatile`) 10
2.1.3 Compliant Solution (`java.util.concurrent.atomic.AtomicBoolean`) 10
2.1.4 Compliant Solution (`synchronized`) 11
2.1.5 Exceptions 12
2.1.6 Risk Assessment 12
2.1.7 References 12
2.2 VNA01-J. Ensure visibility of shared references to immutable objects 13
2.2.1 Noncompliant Code Example 13
2.2.2 Compliant Solution (Synchronization) 14
2.2.3 Compliant Solution (`volatile`) 14
2.2.4 Compliant Solution (`java.util.concurrent` Utilities) 15
2.2.5 Risk Assessment 15
2.2.6 References 15
2.3 VNA02-J. Ensure that compound operations on shared variables are atomic 16
2.3.1 Noncompliant Code Example (Logical Negation) 16
2.3.2 Noncompliant Code Example (Bitwise Negation) 17
2.3.3 Noncompliant Code Example (`volatile`) 17
2.3.4 Compliant Solution (Synchronization) 18
2.3.5 Compliant Solution (Volatile-Read, Synchronized-Write) 18
2.3.6 Compliant Solution (Read-Write Lock) 19
2.3.7 Compliant Solution (`AtomicBoolean`) 20
2.3.8 Noncompliant Code Example (Addition of Primitives) 20
2.3.9 Noncompliant Code Example (Addition of Atomic Integers) 21
2.3.10 Compliant Solution (Addition) 21
2.3.11 Risk Assessment 22
2.3.12 References 22
2.4 VNA03-J. Do not assume that a group of calls to independently atomic methods is atomic 23
2.4.1 Noncompliant Code Example (`AtomicReference`) 23
2.4.2 Compliant Solution (Method Synchronization) 24
2.4.3 Noncompliant Code Example (`synchronizedList`) 24
2.4.4 Compliant Solution (Synchronized Block) 25
2.4.5 Noncompliant Code Example (`synchronizedMap`) 25
2.4.6 Compliant Solution (Synchronization) 26
2.4.7 Compliant Solution (`ConcurrentHashMap`) 26
2.4.8 Risk Assessment 28

2.4.9 References 28
2.5 VNA04-J. Ensure that calls to chained methods are atomic 29
2.5.1 Noncompliant Code Example 29
2.5.2 Compliant Solution 30
2.5.3 Risk Assessment 32
2.5.4 References 32
2.6 VNA05-J. Ensure atomicity when reading and writing 64-bit values 33
2.6.1 Noncompliant Code Example 33
2.6.2 Compliant Solution (Volatile) 33
2.6.3 Exceptions 34
2.6.4 Risk Assessment 34
2.6.5 References 34
2.7 VNA06-J. Do not assume that declaring an object reference volatile guarantees visibility of its members 35
2.7.1 Noncompliant Code Example (Arrays) 35
2.7.2 Compliant Solution (`AtomicIntegerArray`) 36
2.7.3 Compliant Solution (Synchronization) 36
2.7.4 Noncompliant Code Example (Mutable Object) 36
2.7.5 Noncompliant Code Example (Volatile-Read, Synchronized-Write) 37
2.7.6 Compliant Solution (Synchronization) 38
2.7.7 Noncompliant Code Example (Mutable Sub-Object) 39
2.7.8 Compliant Solution (Instance Per Call/Defensive Copying) 39
2.7.9 Compliant Solution (Synchronization) 39
2.7.10 Compliant Solution (`ThreadLocal` Storage) 40
2.7.11 Risk Assessment 40
2.7.12 References 40

3 Lock (LCK) Guidelines 41
3.1 LCK00-J. Use private final lock objects to synchronize classes that may interact with untrusted code 41
3.1.1 Noncompliant Code Example (Method Synchronization) 42
3.1.2 Noncompliant Code Example (Public Non-Final Lock Object) 42
3.1.3 Noncompliant Code Example (Publicly Accessible Non-Final Lock Object) 43
3.1.4 Noncompliant Code Example (Public Final Lock Object) 43
3.1.5 Compliant Solution (Private Final Lock Object) 44
3.1.6 Noncompliant Code Example (Static) 44
3.1.7 Compliant Solution (Static) 45
3.1.8 Exceptions 46
3.1.9 Risk Assessment 46
3.1.10 References 46
3.2 LCK01-J. Do not synchronize on objects that may be reused 47
3.2.1 Noncompliant Code Example (`Boolean` Lock Object) 47
3.2.2 Noncompliant Code Example (Boxed Primitive) 47
3.2.3 Compliant Solution (`Integer`) 48
3.2.4 Noncompliant Code Example (Interned `String` Object) 48
3.2.5 Noncompliant Code Example (`String` Literal) 49
3.2.6 Compliant Solution (`String` Instance) 49
3.2.7 Compliant Solution (Private Final Lock Object) 49
3.2.8 Risk Assessment 50
3.2.9 References 50
3.3 LCK02-J. Do not synchronize on the class object returned by getClass() 51
3.3.1 Noncompliant Code Example (`getClass()` Lock Object) 51
3.3.2 Compliant Solution (Class Name Qualification) 52
3.3.3 Compliant Solution (`Class.forName()`) 52

3.3.4 Noncompliant Code Example (`getClass()` Lock Object, Inner Class) 53
3.3.5 Compliant Solution (Class Name Qualification) 54
3.3.6 Risk Assessment 54
3.3.7 References 54
3.4 LCK03-J. Do not synchronize on the intrinsic locks of high-level concurrency objects 55
3.4.1 Noncompliant Code Example (`ReentrantLock` Lock Object) 55
3.4.2 Compliant Solution (`lock()` and `unlock()`) 55
3.4.3 Risk Assessment 56
3.4.4 References 56
3.5 LCK04-J. Do not synchronize on a collection view if the backing collection is accessible 57
3.5.1 Noncompliant Code Example (Collection View) 57
3.5.2 Compliant Solution (Collection Lock Object) 58
3.5.3 Risk Assessment 58
3.5.4 References 58
3.6 LCK05-J. Synchronize access to static fields that may be modified by untrusted code 59
3.6.1 Noncompliant Code Example 59
3.6.2 Compliant Solution 60
3.6.3 Risk Assessment 60
3.6.4 References 60
3.7 LCK06-J. Do not use an instance lock to protect shared static data 61
3.7.1 Noncompliant Code Example (Non-Static Lock Object for Static Data) 61
3.7.2 Noncompliant Code Example (Method Synchronization for Static Data) 62
3.7.3 Compliant Solution (Static Lock Object) 62
3.7.4 Risk Assessment 62
3.7.5 References 62
3.8 LCK07-J. Avoid deadlock by requesting and releasing locks in the same order 63
3.8.1 Noncompliant Code Example (Different Lock Orders) 63
3.8.2 Compliant Solution (Private Static Final Lock Object) 64
3.8.3 Compliant Solution (Ordered Locks) 65
3.8.4 Compliant Solution (`ReentrantLock`) 67
3.8.5 Noncompliant Code Example (Different Lock Orders, Recursive) 68
3.8.6 Compliant Solution 71
3.8.7 Risk Assessment 72
3.8.8 References 72
3.9 LCK08-J. Ensure actively held locks are released on exceptional conditions 73
3.9.1 Noncompliant Code Example (Checked Exception) 73
3.9.2 Compliant Solution (`finally` Block) 73
3.9.3 Compliant Solution (Execute-Around Idiom) 74
3.9.4 Noncompliant Code Example (Unchecked Exception) 75
3.9.5 Compliant Solution (`finally` Block) 76
3.9.6 Risk Assessment 76
3.9.7 References 76
3.10 LCK09-J. Do not perform operations that may block while holding a lock 77
3.10.1 Noncompliant Code Example (Deferring a Thread) 77
3.10.2 Compliant Solution (Intrinsic Lock) 77
3.10.3 Noncompliant Code Example (Network I/O) 78
3.10.4 Compliant Solution 79
3.10.5 Exceptions 80
3.10.6 Risk Assessment 80
3.10.7 References 80
3.11 LCK10-J. Do not use incorrect forms of the double-checked locking idiom 81
3.11.1 Noncompliant Code Example 82
3.11.2 Compliant Solution (Volatile) 82

3.11.3 Compliant Solution (Static Initialization) 83
3.11.4 Compliant Solution (Initialize-On-Demand, Holder Class Idiom) 83
3.11.5 Compliant Solution (`ThreadLocal` Storage) 84
3.11.6 Compliant Solution (Immutable) 84
3.11.7 Exceptions 85
3.11.8 Risk Assessment 85
3.11.9 References 85
3.12 LCK11-J. Avoid client-side locking when using classes that do not commit to their locking strategy 86
3.12.1 Noncompliant Code Example (Intrinsic Lock) 86
3.12.2 Compliant Solution (Private Final Lock Object) 88
3.12.3 Noncompliant Code Example (Class Extension and Accessible Member Lock) 88
3.12.4 Compliant Solution (Composition) 89
3.12.5 Risk Assessment 90
3.12.6 References 90

4 Thread APIs (THI) Guidelines 91
4.1 THI00-J. Do not assume that the sleep(), yield(), or getState() methods provide synchronization semantics 91
4.1.1 Noncompliant Code Example (`sleep()`) 91
4.1.2 Compliant Solution (Volatile Flag) 92
4.1.3 Compliant Solution (`Thread.interrupt()`) 92
4.1.4 Noncompliant Code Example (`getState()`) 92
4.1.5 Compliant Solution 94
4.1.6 Risk Assessment 94
4.1.7 References 94
4.2 THI01-J. Do not invoke ThreadGroup methods 95
4.2.1 Noncompliant Code Example 95
4.2.2 Compliant Solution 97
4.2.3 Risk Assessment 98
4.2.4 References 98
4.3 THI02-J. Do not invoke Thread.run() 99
4.3.1 Noncompliant Code Example 99
4.3.2 Compliant Solution 99
4.3.3 Exceptions 100
4.3.4 Risk Assessment 100
4.3.5 References 100
4.4 THI03-J. Always invoke wait() and await() methods inside a loop 101
4.4.1 Noncompliant Code Example 102
4.4.2 Compliant Solution 103
4.4.3 Risk Assessment 103
4.4.4 References 103
4.5 THI04-J. Notify all waiting threads instead of a single thread 104
4.5.1 Noncompliant Code Example (`notify()`) 105
4.5.2 Compliant Solution (`notifyAll()`) 106
4.5.3 Noncompliant Code Example (`Condition` interface) 106
4.5.4 Compliant Solution (`signalAll()`) 107
4.5.5 Compliant Solution (Unique Condition Per Thread) 108
4.5.6 Risk Assessment 109
4.5.7 References 109
4.6 THI05-J. Do not use Thread.stop() to terminate threads 110
4.6.1 Noncompliant Code Example (Deprecated `Thread.stop()`) 110
4.6.2 Compliant Solution (Volatile Flag) 111
4.6.3 Compliant Solution (Interruptible) 112

4.6.4 Compliant Solution (Runtime Permission `stopThread`) 113
4.6.5 Risk Assessment 113
4.6.6 References 113
4.7 THI06-J. Ensure that threads and tasks performing blocking operations can be terminated 114
4.7.1 Noncompliant Code Example (Blocking I/O, Volatile Flag) 114
4.7.2 Noncompliant Code Example (Blocking I/O, Interruptible) 115
4.7.3 Compliant Solution (Close Socket Connection) 115
4.7.4 Compliant Solution (Interruptible Channel) 116
4.7.5 Noncompliant Code Example (Database Connection) 117
4.7.6 Compliant Solution (`Statement.cancel()`) 118
4.7.7 Risk Assessment 120
4.7.8 References 120

5 Thread Pools (TPS) Guidelines 121
5.1 TPS00-J. Use thread pools to enable graceful degradation of service during traffic bursts 121
5.1.1 Noncompliant Code Example 121
5.1.2 Compliant Solution 122
5.1.3 Risk Assessment 124
5.1.4 References 124
5.2 TPS01-J. Do not execute interdependent tasks in a bounded thread pool 125
5.2.1 Noncompliant Code Example (Interdependent Subtasks) 125
5.2.2 Compliant Solution (No Interdependent Tasks) 127
5.2.3 Noncompliant Code Example (Subtasks) 128
5.2.4 Compliant Solution (`CallerRunsPolicy`) 130
5.2.5 Risk Assessment 131
5.2.6 References 131
5.3 TPS02-J. Ensure that tasks submitted to a thread pool are interruptible 132
5.3.1 Noncompliant Code Example (Shutting Down Thread Pools) 132
5.3.2 Compliant Solution (Submit Interruptible Tasks) 133
5.3.3 Exceptions 134
5.3.4 Risk Assessment 134
5.3.5 References 134
5.4 TPS03-J. Ensure that tasks executing in a thread pool do not fail silently 135
5.4.1 Noncompliant Code Example (Abnormal Task Termination) 135
5.4.2 Compliant Solution (`ThreadPoolExecutor` Hooks) 135
5.4.3 Compliant Solution (Uncaught Exception Handler) 136
5.4.4 Compliant Solution (`Future<V>` and `submit()`) 138
5.4.5 Exceptions 138
5.4.6 Risk Assessment 138
5.4.7 References 138
5.5 TPS04-J. Ensure ThreadLocal variables are reinitialized when using thread pools 139
5.5.1 Noncompliant Code Example 139
5.5.2 Noncompliant Code Example (Increase Thread Pool Size) 141
5.5.3 Compliant Solution (`try-finally` Clause) 141
5.5.4 Compliant Solution (`beforeExecute()`) 142
5.5.5 Exceptions 143
5.5.6 Risk Assessment 143
5.5.7 References 143

6 Thread-Safety Miscellaneous (TSM) Guidelines 145
6.1 TSM00-J. Do not override thread-safe methods with methods that are not thread-safe 145
6.1.1 Noncompliant Code Example (Synchronized Method) 145

6.1.2 Compliant Solution (Synchronized Method) 146
6.1.3 Compliant Solution (Private Final Lock Object) 146
6.1.4 Noncompliant Code Example (Private Lock) 147
6.1.5 Compliant Solution (Private Lock) 147
6.1.6 Risk Assessment 148
6.1.7 References 148
6.2 TSM01-J. Do not let the “this” reference escape during object construction 149
6.2.1 Noncompliant Code Example (Publish Before Initialization) 150
6.2.2 Noncompliant Code Example (Non-Volatile Public Static Field) 150
6.2.3 Compliant Solution (Volatile Field and Publish After Initialization) 151
6.2.4 Compliant Solution (Public Static Factory Method) 151
6.2.5 Noncompliant Code Example (Handlers) 152
6.2.6 Compliant Solution 153
6.2.7 Noncompliant Code Example (Inner Class) 154
6.2.8 Compliant Solution 154
6.2.9 Noncompliant Code Example (Thread) 155
6.2.10 Compliant Solution (Thread) 155
6.2.11 Exceptions 156
6.2.12 Risk Assessment 156
6.2.13 References 156
6.3 TSM02-J. Do not use background threads during class initialization 157
6.3.1 Noncompliant Code Example (Background Thread) 157
6.3.2 Compliant Solution (`static` Initializer, No Background Threads) 158
6.3.3 Compliant Solution (`ThreadLocal`) 159
6.3.4 Exceptions 160
6.3.5 Risk Assessment 161
6.3.6 References 161
6.4 TSM03-J. Do not publish partially initialized objects 162
6.4.1 Noncompliant Code Example 162
6.4.2 Compliant Solution (Synchronization) 163
6.4.3 Compliant Solution (Final Field) 164
6.4.4 Compliant Solution (Final Field and Thread-Safe Composition) 164
6.4.5 Compliant Solution (Static Initialization) 165
6.4.6 Compliant Solution (Immutable Object - Final Fields, Volatile Reference) 166
6.4.7 Compliant Solution (Mutable Thread-Safe Object, Volatile Reference) 166
6.4.8 Exceptions 168
6.4.9 Risk Assessment 168
6.4.10 References 168
6.5 TSM04-J. Document thread-safety and use annotations where applicable 169
6.5.1 Obtaining Concurrency Annotations 169
6.5.2 Documenting Intended Thread-Safety 169
6.5.3 Documenting Locking Policies 171
6.5.4 Construction of Mutable Objects 173
6.5.5 Documenting Thread-Confinement Policies 173
6.5.6 Documenting Wait-Notify Protocols 174
6.5.7 Risk Assessment 174
6.5.8 References 174

Appendix Definitions 175

Bibliography 181

List of Figures

Figure 1:	Guideline Priorities	xiv
Figure 2:	Modern, Shared-Memory, Multiprocessor Architecture	1
Figure 3:	Example Threads and Their Executing Statements	4
Figure 4:	How Backing Collection Works in the Collection View, Noncompliant Code Example	57

List of Tables

Table 1:	Example Thread Assignment #1	2
Table 2:	Example #1 of Assignments in Order of Execution	2
Table 3:	Example #2 of Assignments in Order of Execution	2
Table 4:	Possible Reorderings Between Volatile and Non-Volatile Variables	5
Table 5:	Example Thread Assignment #2	5
Table 6:	Execution Order #1	5
Table 7:	Execution Order #2	5

Acknowledgments

We want to thank everyone who contributed their time and effort to the development of these guidelines, including Siddarth Adukia, Lokesh Agarwal, Ron Bandes, Kalpana Chatnani, Jose Sandoval Chaverri, Tim Halloran (SureLogic), Thomas Hawtin, Fei He, Ryan Hofler, Sam Kaplan, Georgios Katsis, Lothar Kimmeringer, Bastian Marquis, Michael Kross, Christopher Leonavicius, Bocong Liu, Efstathios Mertikas, David Neville, Justin Pincar, Michael Rosenman, Eric Schwelm, Tamir Sen, Philip Shirey, Jagadish Shrinivasavadhani, Robin Steiger, John Truelove, Theti Tsiampali, Tim Wilson, and Weam Abu Zaki.

We also want to thank the following people for their careful reviews of both this technical report and the wiki on which it is based: Hans Boehm, Joseph Bowbeer, Klaus Havelund, David Holmes, Bart Jacobs, Niklas Matthies, Bill Michell, Philip Miller, Nick Morrott, Attila Mravik, Tim Peierls, Alex Snaps, Kenneth A. Williams.

We also thank our editors: Pamela Curtis and Pennie Walters.

This research was supported by the U.S. Department of Defense (DoD) and the U.S. Department of Homeland Security (DHS) National Cyber Security Division (NCSD).

About This Report

The Secure Coding Standard Described in This Report

The CERT Oracle Secure Coding Standard for Java is the result of a collaboration between the CERT® Program at the Carnegie Mellon® Software Engineering Institute and Oracle. It is being developed as a community effort on the CERT secure coding wiki located at www.securecoding.cert.org. This report contains a subset of those guidelines that deal with concurrency and may undergo further revision before being published as part of the CERT Oracle Secure Coding Standard for Java. The concurrency guidelines are divided into the following categories:

- visibility and atomicity (VNA)
- locks (LCK)
- thread APIs (THI)
- thread pools (TPS)
- thread-safety miscellaneous (TSM)

We welcome your feedback about these guidelines. To comment on the wiki, simply go to it and sign up for a wiki account.

Guideline Priorities

Each guideline has a priority assigned using a metric based on Failure Mode, Effects, and Criticality Analysis (FMECA) [IEC 2006]. A value for each of the following is assigned to each guideline:

- severity – If the guideline is ignored, how serious are the consequences?
  1 = low (denial-of-service attack, abnormal termination)
  2 = medium (data integrity violation, unintentional information disclosure)
  3 = high (run arbitrary code, privilege escalation)
- likelihood – If the guideline is ignored and that results in the introduction of a flaw, how likely is it for that flaw to lead to an exploitable vulnerability?
  1 = unlikely
  2 = probable
  3 = likely
- remediation cost – How expensive is it to comply with the guideline?
  1 = high (manual detection and correction)
  2 = medium (automatic detection and manual correction)
  3 = low (automatic detection and correction)

The three values are then multiplied for each guideline. The resulting value, which will be between 1 and 27, provides a measure that can be used to prioritize the application of the guidelines.

Guidelines with a priority in the range of 1-4 are level-3 guidelines; those in the range of 6-9 are level-2; and those in the range of 12-27 are level-1. As a result, it is possible to claim level-1, level-2, or complete compliance (level-3) with a standard by implementing all guidelines in a level, as shown in Figure 1.

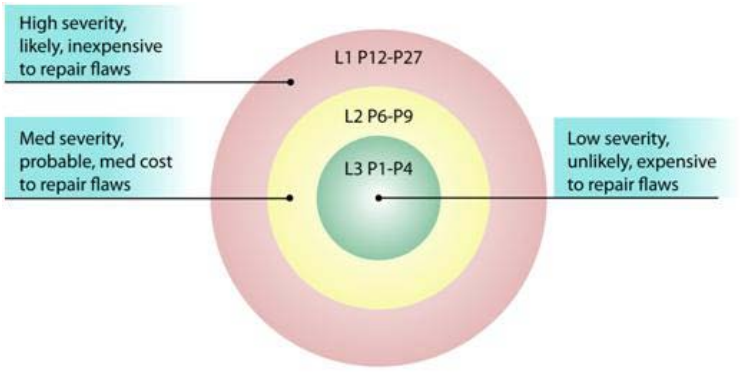

Figure 1: Guideline Priorities

This metric is designed primarily for remediation projects. New development efforts are expected to conform to the entire standard.

Abstract

An essential element of secure coding in the Java programming language is well-documented and enforceable coding standards. Coding standards encourage programmers to follow a uniform set of guidelines determined by the requirements of the project and organization, rather than by the programmer's familiarity or preference. Once established, these standards can be used as a metric to evaluate source code (using manual or automated processes).

The CERT Oracle Secure Coding Standard for Java provides guidelines for secure coding in the Java programming language. The goal of these guidelines is to eliminate insecure coding practices and undefined behaviors that can lead to exploitable vulnerabilities. Applying this standard will lead to higher quality systems that are robust and more resistant to attack.

This report documents the portion of those Java guidelines that are related to concurrency.

1 Introduction

Memory that can be shared between threads is called *shared memory* or *heap memory*. The term *variable* as used in this technical report refers to both fields and array elements [Gosling 2005]. Variables that are shared between threads are referred to as *shared variables*. All instance fields, static fields, and array elements are shared variables allocated in heap memory. Local variables, formal method parameters, and exception-handler parameters are never shared between threads and are not affected by the memory model.

In a modern, shared-memory, multiprocessor architecture, each processor has one or more levels of cache that are periodically reconciled with main memory as shown in Figure 2.

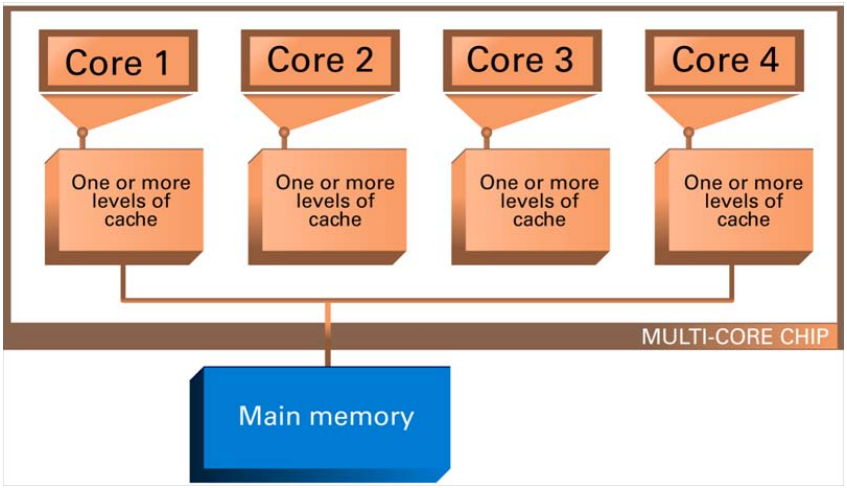

Figure 2: Modern, Shared-Memory, Multiprocessor Architecture

The visibility of writes to shared variables can be problematic because the value of a shared variable may be cached and not written to main memory immediately. Consequently, another thread may read a stale value of the variable.

A further concern is that concurrent executions of code are typically interleaved, and statements may be reordered by the compiler or runtime system to optimize performance. This results in execution orders that are not immediately obvious when the source code is examined. Failure to account for possible reorderings is a common source of data races.

Consider the following example in which `a` and `b` are (shared) global variables or instance fields, but `r1` and `r2` are local variables that are not accessible to other threads.

Initially, let `a = 0` and `b = 0`, as shown in Table 1:

Table 1: Example Thread Assignment #1

Thread 1	Thread 2
a = 10;	b = 20;
r1 = b;	r2 = a;

Because the two assignments in Thread 1 (`a = 10;` and `r1 = b;`) are unrelated, the compiler or runtime system is free to reorder them. Similarly in Thread 2, the statements may be reordered freely. Although it may seem counterintuitive, the Java Memory Model (JMM) allows a read to see the value of a write that occurs later in the execution order.

A possible execution order showing actual assignments is shown in Table 2.

Table 2: Example #1 of Assignments in Order of Execution

Execution Order (Time)	Thread#	Assignment	Assigned Value	Notes
1	t_1	a = 10;	10	
2	t_2	b = 20;	20	
3	t_1	r1 = b;	0	Reads initial value of b, that is 0
4	t_2	r2 = a;	0	Reads initial value of a, that is 0

In this ordering, `r1` and `r2` read the original values of variables `b` and `a`, respectively, even though they are expected to see the updated values, 20 and 10. Another possible execution order showing actual assignments is shown in Table 3.

Table 3: Example #2 of Assignments in Order of Execution

Execution Order (Time)	Thread#	Assignment	Assigned Value	Notes
1	t_1	r1 = b;	20	Reads later value (in Step 4) of write, that is 20
2	t_2	r2 = a;	10	Reads later value (in Step 3) of write, that is 10
3	t_1	a = 10;	10	
4	t_2	b = 20;	20	

In this ordering, `r1` and `r2` read the values of `a` and `b` written from Steps 3 and 4, even before the statements corresponding to these steps have executed.

Restricting the set of possible reorderings makes it easier to reason about the correctness of the code.

Even if statements execute in the order of their appearance in a thread, caching can prevent the latest values from being reflected in the main memory.

The *Java Language Specification* defines the JMM that provides certain guarantees to the Java programmer. The JMM is specified in terms of actions, which include variable reads and writes;

monitor locks and unlocks; and thread starts and joins. The JMM defines a partial ordering called *happens-before* on all actions within the program. To guarantee that a thread executing action B can see the results of action A, for example, a happens-before relationship must be defined such that A happens-before B.

According to Section 17.4.5 "Happens-before Order" of the *Java Language Specification* [Gosling 2005]

1. *An unlock on a monitor happens-before every subsequent lock on that monitor.*
2. *A write to a volatile field happens-before every subsequent read of that field.*
3. *A call to start() on a thread happens-before any actions in the started thread.*
4. *All actions in a thread happens-before any other thread successfully returns from a join() on that thread.*
5. *The default initialization of any object happens-before any other actions (other than default-writes) of a program.*
6. *A thread calling interrupt on another thread happens-before the interrupted thread detects the interrupt.*
7. *The end of a constructor for an object happens-before the start of the finalizer for that object.*

If a happens-before relationship does not exist between two operations, the Java Virtual Machine (JVM) is free to reorder them. A data race occurs when a variable is written to by at least one thread and read by at least another thread, and the reads and writes are not ordered by a happens-before relationship. A correctly synchronized program is one with no data races. The JMM guarantees *sequential consistency* for correctly synchronized programs. Sequential consistency means that the result of any execution is the same as if the reads and writes by all threads on shared data were executed in some sequential order and the operations of each individual thread appear in this sequence in the order specified by its program [Tanenbaum 2002]—in other words

1. Take the read and write operations performed by each thread and put them in the order in which the thread executes them (thread order).
2. Interleave the operations in some way allowed by the happens-before relationships to form an execution order.
3. Read operations must return the most recently written data in the total program order for the execution to be sequentially consistent.

If the program is sequentially consistent, all threads see the same total ordering of reads and writes of shared variables.

The actual execution order of instructions and memory accesses can vary as long as

- the actions of the thread appear to that thread as if program order were followed
- all values read are allowed for by the memory model

These constraints allow the programmer to understand the semantics of the programs they write and allow compiler writers and virtual machine implementers to perform various optimizations [Arnold 2006].

Several concurrency primitives can help a programmer reason about the semantics of multi-threaded programs.

1.1.1 The `volatile` Keyword

Declaring shared variables volatile ensures visibility and limits reordering of accesses. Volatile accesses do not guarantee the atomicity of composite operations such as incrementing a variable. Consequently, `volatile` is not applicable in cases where the atomicity of composite operations must be guaranteed. (See guideline "VNA02-J. Ensure that compound operations on shared variables are atomic" on page 16 for more information.)

Declaring variables volatile establishes a happens-before relationship such that a write to a volatile variable is always seen by threads performing subsequent reads of the same variable. Statements that occur before the write to the volatile field also happens-before any reads of the volatile field.

Consider two threads that are executing some statements as shown in Figure 3Figure 1.

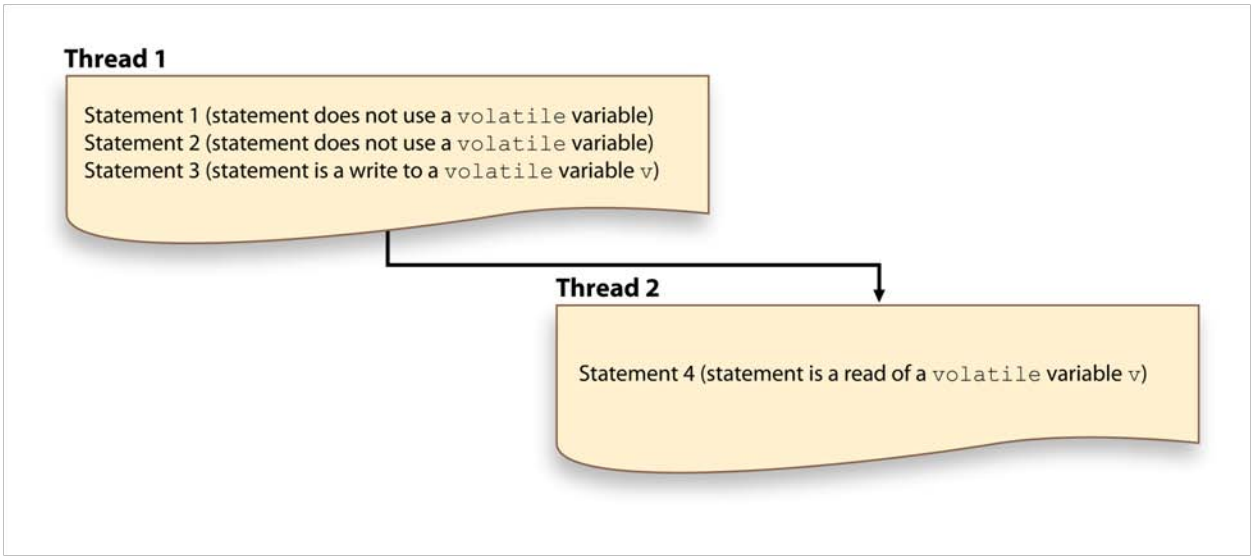

Figure 3: Example Threads and Their Executing Statements

Thread 1 and Thread 2 have a happens-before relationship such that Thread 2 does not start before Thread 1 finishes.

In this example, statement 3 writes to a volatile variable, and statement 4 (in Thread 2) reads the same volatile variable. The read sees the most recent write (to the same variable `v`) from statement 3.

Volatile read and write operations cannot be reordered with respect to each other or to non-volatile variables accesses. When Thread 2 reads the volatile variable, it sees the results of all the writes occurring before the write to the volatile variable in Thread 1. Because of the relatively strong guarantees of volatile read and write operations, the performance overhead is almost the same as that of synchronization.

In the previous example, there is no guarantee that statements 1 and 2 will be executed in the order in which they appear in the program. They may be reordered freely by the compiler because there is no happens-before relationship between these two statements.

The possible reorderings between volatile and non-volatile variables are summarized in Table 4. Load and store operations are synonymous to read and write operations, respectively [Lea 2008].

Table 4: Possible Reorderings Between Volatile and Non-Volatile Variables

Can Reorder	***2nd Operation***			
1st Operation	**Normal Load**	**Normal Store**	**Volatile Load**	**Volatile Store**
Normal load	Yes	Yes	Yes	No
Normal store	Yes	Yes	Yes	No
Volatile load	No	No	No	No
Volatile store	Yes	Yes	No	No

1.1.2 Synchronization

A correctly synchronized program is one whose sequentially consistent executions do not have any data races. The example shown below uses a non-volatile variable x and a volatile variable y and is not correctly synchronized.

Table 5: Example Thread Assignment #2

Thread 1	**Thread 2**
x = 1	r1 = y
y = 2	r2 = x

The two sequentially consistent execution orders of this example are shown in Table 6 and Table 7.

Table 6: Execution Order #1

Step (Time)	**Thread#**	**Statement**	**Comment**
1	t_1	x = 1	Write to non-volatile variable
2	t_1	y = 2	Write to volatile variable
3	t_2	r1 = y	Read of volatile variable
4	t_2	r2 = x	Read of non-volatile variable

Table 7: Execution Order #2

Step (Time)	**Thread#**	**Statement**	**Comment**
1	t_2	r1 = y	Read of volatile variable
2	t_2	r2 = x	Read of non-volatile variable
3	t_1	x = 1	Write to non-volatile variable
4	t_1	y = 2	Write to volatile variable

In the first case, a happens-before relationship exists between actions such that Steps 1 and 2 always occur before Steps 3 and 4. However, in the second case, no happens-before relationship exists between any of the steps. Consequently, because there is a sequentially consistent execution that has no happens-before relationship, this example contains data races.

Correct visibility guarantees that multiple threads accessing shared data can view each others' results, but does not establish the order of when each thread reads or writes the data. Correct synchronization guarantees that threads access data in a proper order. For example, the code shown below ensures that there is only one sequentially consistent execution order that performs all the actions of Thread 1 before Thread 2.

```
class Assign {
  public synchronized void doSomething() {
    // Perform Thread 1 actions
    x = 1;
    y = 2;
    // Perform Thread 2 actions
    r1 = y;
    r2 = x;
  }
}
```

When using synchronization, there is no need to declare the variable `y` volatile. Synchronization involves acquiring a lock, performing operations, and then releasing the lock. In the above example, the `doSomething()` method acquires the intrinsic lock of the class object (`Assign`). This example can also be written to use block synchronization:

```
class Assign {
  public void doSomething() {
    synchronized (this) {
      // Perform Thread 1 actions
      x = 1;
      y = 2;
      // Perform Thread 2 actions
      r1 = y;
      r2 = x;
    }
  }
}
```

The intrinsic lock used in both examples is the same.

1.1.3 The `java.util.concurrent` Classes

1.1.3.1 Atomic Classes

Volatile variables are useful for guaranteeing visibility. However, they are insufficient for ensuring atomicity. Synchronization fills this gap but incurs overheads of context switching and frequently causes lock contention. The atomic classes of package `java.util.concurrent.atomic` provide a mechanism for reducing contention in most practical environments while at the same time ensuring atomicity. According to Goetz and colleagues [Goetz 2006]

> *With low to moderate contention, atomics offer better scalability; with high contention, locks offer better contention avoidance.*

The atomic classes consist of implementations that exploit the design of modern processors by exposing commonly needed functionality to the programmer. For example, the `AtomicInteger.incrementAndGet()` method can be used for atomically incrementing a variable. The compare-and-swap instruction(s) provided by modern processors offer more fine-grained control and can be used directly by invoking high-level methods such as

`java.util.concurrent.atomic.Atomic*.compareAndSet()` where the asterisk can be, for example, an `Integer`, `Long`, or `Boolean`.

1.1.3.2 The `Executor` Framework

The `java.util.concurrent` package provides the `Executor` framework that offers a mechanism for executing tasks concurrently. A *task* is a logical unit of work encapsulated by a class that implements `Runnable` or `Callable`. The `Executor` framework allows task submission to be decoupled from low-level scheduling and thread management details. It provides the thread pool mechanism that allows a system to degrade gracefully when presented with more requests than the system can handle simultaneously.

The `Executor` interface is the core interface of the framework and is extended by the `ExecutorService` interface that provides facilities for thread pool termination and obtaining return values of tasks (Futures). The `ExecutorService` interface is further extended by the `ScheduledExecutorService` interface that provides a way to run tasks periodically or after some delay. The `Executors` class provides several factory and utility methods that are preconfigured with commonly used configurations of `Executor`, `ExecutorService`, and other related interfaces. For example, the `Executors.newFixedThreadPool()` method returns a fixed size thread pool with an upper limit on the number of concurrently executing tasks and maintains an unbounded queue for holding tasks while the thread pool is full. The base (actual) implementation of the thread pool is provided by the `ThreadPoolExecutor` class. This class can be instantiated to customize the task execution policy.

The `java.util.concurrent` utilities are preferred over traditional synchronization primitives such as synchronization and volatile variables because the `java.util.concurrent` utilities abstract the underlying details, provide a cleaner and less error-prone API, are easier to scale, and can be enforced using policies.

1.1.3.3 Explicit Locking

The `java.util.concurrent` package provides the `ReentrantLock` class that has additional features not provided by intrinsic locks. For example, the `ReentrantLock.tryLock()` method does not block waiting if another thread is already holding the lock. Acquiring and releasing a `ReentrantLock` has the same semantics as acquiring and releasing an intrinsic lock.

2 Visibility and Atomicity (VNA) Guidelines

2.1 VNA00-J. Ensure visibility when accessing shared primitive variables

Reading a shared primitive variable in one thread may not yield the value of the most recent write to the variable from another thread. Consequently, the thread may observe a stale value of the shared variable. To ensure the visibility of the most recent update, either the variable must be declared volatile or the reads and writes must be synchronized.

Declaring a shared variable volatile guarantees visibility in a thread-safe manner only when both of the following conditions are met:

- A write to a variable does not depend on its current value.
- A write to a variable does not depend on the result of any non-atomic compound operations involving reads and writes of other variables. (For more information, see guideline “VNA02-J. Ensure that compound operations on shared variables are atomic” on page 16.)

The first condition can be relaxed when you can be sure that only one thread will ever update the value of the variable [Goetz 2006]. However, code that relies on a single-thread confinement is error-prone and difficult to maintain. This behavior is permissible under this guideline but not recommended.

Synchronizing the code makes it easier to reason about its behavior and is frequently more secure than simply using the `volatile` keyword. However, synchronization has a somewhat higher performance overhead and can result in thread contention and deadlocks when used excessively.

Declaring a variable volatile or correctly synchronizing the code guarantees that 64-bit primitive `long` and `double` variables are accessed atomically. (For more information on sharing those variables among multiple threads, see guideline “VNA05-J. Ensure atomicity when reading and writing 64-bit values” on page 33.)

2.1.1 Noncompliant Code Example (Non-Volatile Flag)

This noncompliant code example uses a `shutdown()` method to set a non-volatile `done` flag that is checked in the `run()` method.

```
final class ControlledStop implements Runnable {
  private boolean done = false;

  @Override public void run() {
    while (!done) {
      try {
        // ...
        Thread.currentThread().sleep(1000); // Do something
      } catch(InterruptedException ie) {
        Thread.currentThread().interrupt(); // Reset interrupted status
      }
```

```
    }
  }

  public void shutdown() {
    done = true;
  }
}
```

If one thread invokes the `shutdown()` method to set the flag, a second thread might not observe that change. Consequently, the second thread may observe that `done` is still false and incorrectly invoke the `sleep()` method. A compiler is allowed to optimize the code if it determines that the value of `done` is never modified by the same thread, resulting in an infinite loop.

2.1.2 Compliant Solution (`volatile`)

In this compliant solution, the `done` flag is declared volatile to ensure that writes are visible to other threads.

```
final class ControlledStop implements Runnable {
  private volatile boolean done = false;

  @Override public void run() {
    while (!done) {
      try {
        // ...
        Thread.currentThread().sleep(1000); // Do something
      } catch(InterruptedException ie) {
        Thread.currentThread().interrupt(); // Reset interrupted status
      }
    }
  }

  public void shutdown() {
    done = true;
  }
}
```

2.1.3 Compliant Solution (`java.util.concurrent.atomic.AtomicBoolean`)

In this compliant solution, the `done` flag is declared `AtomicBoolean`. Atomic types also guarantee that writes are visible to other threads.

```
final class ControlledStop implements Runnable {
  private final AtomicBoolean done = new AtomicBoolean(false);

  @Override public void run() {
    while (!done.get()) {
      try {
        // ...
        Thread.currentThread().sleep(1000); // Do something
```

```
      } catch(InterruptedException ie) {
        Thread.currentThread().interrupt(); // Reset interrupted status
      }
    }
  }

  public void shutdown() {
    done.set(true);
  }
}
```

2.1.4 Compliant Solution (synchronized)

This compliant solution uses the intrinsic lock of the `Class` object to ensure that updates become visible to other threads.

```
final class ControlledStop implements Runnable {
  private boolean done = false;

  @Override public void run() {
    while (!isDone()) {
      try {
        // ...
        Thread.currentThread().sleep(1000); // Do something
      } catch(InterruptedException ie) {
        Thread.currentThread().interrupt(); // Reset interrupted status
      }
    }
  }

  public synchronized boolean isDone() {
    return done;
  }

  public synchronized void shutdown() {
    done = true;
  }
}
```

While this is an acceptable compliant solution, intrinsic locks cause threads to block and may introduce contention. On the other hand, volatile-qualified shared variables do not block. Excessive synchronization can also make the program prone to deadlock.

Synchronization is a more secure alternative in situations where the `volatile` keyword or a `java.util.concurrent.atomic.Atomic*` field is inappropriate, such as if a variable's new value depends on its current value. For more information, see guideline "VNA02-J. Ensure that compound operations on shared variables are atomic" on page 16.

Compliance with guideline “LCK00-J. Use private final lock objects to synchronize classes that may interact with untrusted code” on page 41 can reduce the likelihood of misuse by ensuring that untrusted callers cannot access the lock object.

2.1.5 Exceptions

VNA00-EX1: `Class` objects need not be made visible because they are created by the virtual machine and their initialization always precedes any subsequent use.

2.1.6 Risk Assessment

Failing to ensure the visibility of a shared primitive variable may result in a thread observing a stale value of the variable.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
VNA00-J	medium	probable	medium	P8	L2

2.1.7 References

[Arnold 2006] Section 14.10.3, “The Happens-Before Relationship”

[Bloch 2008] Item 66: “Synchronize access to shared mutable data”

[Gosling 2005] Chapter 17, Threads and Locks:
Section 17.4.5, “Happens-Before Order”
Section 17.4.3, “Programs and Program Order”
Section 17.4.8, “Executions and Causality Requirements”

[MITRE 2010] CWE ID 667, “Insufficient Locking”
CWE ID 413, “Insufficient Resource Locking”
CWE ID 567, “Unsynchronized Access to Shared Data”

2.2 VNA01-J. Ensure visibility of shared references to immutable objects

A common misconception is that shared references to immutable objects are visible across multiple threads as soon as they are updated. For example, a developer can mistakenly believe that a class containing fields referring to only immutable objects is immutable and, consequently, thread-safe.

Section 14.10.2, "Final Fields and Security" of *Java Programming Language, Fourth Edition* states [Arnold 2006]

> *The problem is that, while the shared object is immutable, the reference used to access the shared object is itself shared and often mutable. Consequently, a correctly synchronized program must synchronize access to that shared reference, but often programs do not do this, because programmers do not recognize the need to do it.*

References to both immutable and mutable objects must be made visible to all the threads. Immutable objects can be shared safely among multiple threads. However, mutable objects may not be fully constructed when their references are made visible. Guideline "TSM03-J. Do not publish partially initialized objects" on page 162 describes object construction and visibility issues specific to mutable objects.

2.2.1 Noncompliant Code Example

This noncompliant code example consists of the immutable `Helper` class:

```
// Immutable Helper
public final class Helper {
  private final int n;

  public Helper(int n) {
    this.n = n;
  }
  // ...
}
```

and a mutable `Foo` class:

```
final class Foo {
  private Helper helper;

  public Helper getHelper() {
    return helper;
  }

  public void setHelper(int num) {
    helper = new Helper(num);
  }
}
```

The `getHelper()` method publishes the mutable `helper` field. Because the `Helper` class is immutable, it cannot be changed after it is initialized. Furthermore, because `Helper` is immutable, it is always constructed properly before its reference is made visible in compliance with guideline "TSM03-J. Do not publish partially initialized objects" on page 162. Unfortunately, a separate thread could observe a stale reference in the `helper` field of the `Foo` class.

2.2.2 Compliant Solution (Synchronization)

This compliant solution synchronizes the methods of the `Foo` class to ensure that no thread sees a stale `Helper` reference.

```
final class Foo {
  private Helper helper;

  public synchronized Helper getHelper() {
    return helper;
  }

  public synchronized void setHelper(int num) {
    helper = new Helper(num);
  }
}
```

The immutable `Helper` class remains unchanged.

2.2.3 Compliant Solution (`volatile`)

References to immutable member objects can be made visible by declaring them volatile.

```
final class Foo {
  private volatile Helper helper;

  public Helper getHelper() {
    return helper;
  }

  public void setHelper(int num) {
    helper = new Helper(num);
  }
}
```

The immutable `Helper` class remains unchanged.

2.2.4 Compliant Solution (`java.util.concurrent` Utilities)

This compliant solution wraps the immutable `Helper` object within an `AtomicReference` wrapper that can be updated atomically.

```
final class Foo {
  private final AtomicReference<Helper> helperRef =
    new AtomicReference<Helper>();

  public Helper getHelper() {
    return helperRef.get();
  }

  public void setHelper(int num) {
    helperRef.set(new Helper(num));
  }
}
```

The immutable `Helper` class remains unchanged.

2.2.5 Risk Assessment

The assumption that classes containing immutable objects are immutable is incorrect and can cause serious thread-safety issues.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
VNA01-J	low	probable	medium	P4	L3

2.2.6 References

[Arnold 2006] Section 14.10.2, “Final Fields and Security”

[Goetz 2006] Section 3.4.2, “Example: Using Volatile to Publish Immutable Objects”

[Sun 2009b]

2.3 VNA02-J. Ensure that compound operations on shared variables are atomic

Compound operations are operations that consist of more than one discrete operation. Expressions that include postfix or prefix increment (`++`), postfix or prefix decrement (`--`), or compound assignment operators always result in compound operations. Compound assignment expressions use operators such as `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `>>>=`, `^=`, and `|=` [Gosling 2005]. Compound operations on shared variables must be performed atomically to prevent data races and race conditions.

For information about the atomicity of a grouping of calls to independently atomic methods that belong to thread-safe classes, see guideline "VNA03-J. Do not assume that a group of calls to independently atomic methods is atomic" on page 23.

The *Java Language Specification* also permits reads and writes of 64-bit values to be non-atomic. For more information, see guideline "VNA05-J. Ensure atomicity when reading and writing 64-bit values" on page 33.

2.3.1 Noncompliant Code Example (Logical Negation)

This noncompliant code example declares a shared `boolean flag` variable and provides a `toggle()` method that negates the current value of `flag`.

```
final class Flag {
  private boolean flag = true;

  public void toggle() {  // Unsafe
    flag = !flag;
  }

  public boolean getFlag() { // Unsafe
    return flag;
  }
}
```

Execution of this code may result in a data race because the value of `flag` is read, negated, and written back.

Consider, for example, two threads that call `toggle()`. The expected effect of toggling `flag` twice is that it is restored to its original value. However, the following scenario leaves `flag` in the incorrect state:

Time	flag=	Thread	Action
1	true	t_1	reads the current value of `flag`, true, into a temporary variable
2	true	t_2	reads the current value of `flag`, (still) true, into a temporary variable
3	true	t_1	toggles the temporary variable to false
4	true	t_2	toggles the temporary variable to false
5	false	t_1	writes the temporary variable's value to `flag`
6	false	t_2	writes the temporary variable's value to `flag`

As a result, the effect of the call by t_2 is not reflected in `flag`; the program behaves as if `toggle()` was called only once, not twice.

2.3.2 Noncompliant Code Example (Bitwise Negation)

Similarly, the `toggle()` method can use the compound assignment operator `^=` to negate the current value of `flag`.

```
final class Flag {
  private boolean flag = true;

  public void toggle() {  // Unsafe
    flag ^= true;  // Same as flag = !flag;
  }

  public boolean getFlag() { // Unsafe
    return flag;
  }
}
```

This code is also not thread-safe. A data race exists because `^=` is a non-atomic compound operation.

2.3.3 Noncompliant Code Example (`volatile`)

Declaring `flag` volatile does not help either:

```
final class Flag {
  private volatile boolean flag = true;

  public void toggle() {  // Unsafe
    flag ^= true;
  }

  public boolean getFlag() { // Safe
```

```
    return flag;
  }
}
```

This code remains unsuitable for multithreaded use because declaring a variable volatile does not guarantee the atomicity of compound operations on it.

2.3.4 Compliant Solution (Synchronization)

This compliant solution declares both the `toggle()` and `getFlag()` methods as synchronized.

```
final class Flag {
  private boolean flag = true;

  public synchronized void toggle() {
    flag ^= true; // Same as flag = !flag;
  }

  public synchronized boolean getFlag() {
    return flag;
  }
}
```

This guards the reads and writes to the `flag` field with a lock on the instance, that is, `this`. This compliant solution ensures that changes are visible to all the threads. Now, only two execution orders are possible, one of which is shown below.

Time	flag=	Thread	Action
1	true	t_1	reads the current value of `flag`, true, into a temporary variable
2	true	t_1	toggles the temporary variable to false
3	false	t_1	writes the temporary variable's value to `flag`
4	false	t_2	reads the current value of `flag`, false, into a temporary variable
5	false	t_2	toggles the temporary variable to true
6	true	t_2	writes the temporary variable's value to `flag`

The second execution order involves the same operations, but t_2 starts and finishes before t_1.

Compliance with guideline “LCK00-J. Use private final lock objects to synchronize classes that may interact with untrusted code” on page 41 can reduce the likelihood of misuse by ensuring that untrusted callers cannot access the lock object.

2.3.5 Compliant Solution (Volatile-Read, Synchronized-Write)

In this compliant solution, the `getFlag()` method is not synchronized, and `flag` is declared volatile. This solution is compliant because the read of `flag` in the `getFlag()` method is an atomic operation and the volatile qualification assures visibility. The `toggle()` method still requires synchronization because it performs a non-atomic operation.

```
final class Flag {
  private volatile boolean flag = true;

  public synchronized void toggle() {
    flag ^= true; // Same as flag = !flag;
  }

  public boolean getFlag() {
    return flag;
  }
}
```

This approach may not be used when a getter method performs operations other than just returning the value of a volatile field without having to use any synchronization. Unless read performance is critical, this technique may not offer significant advantages over synchronization [Goetz 2006].

Guideline "VNA06-J. Do not assume that declaring an object reference volatile guarantees visibility of its members" on page 35 also addresses the volatile-read, synchronized-write pattern.

2.3.6 Compliant Solution (Read-Write Lock)

This compliant solution uses a read-write lock to ensure atomicity and visibility.

```
final class Flag {
  private boolean flag = true;
  private final ReadWriteLock lock = new ReentrantReadWriteLock();
  private final Lock readLock = lock.readLock();
  private final Lock writeLock = lock.writeLock();

  public synchronized void toggle() {
    writeLock.lock();
    try {
      flag ^= true; // Same as flag = !flag;
    } finally {
      writeLock.unlock();
    }
  }

  public boolean getFlag() {
    readLock.lock();
    try {
      return flag;
    } finally {
      readLock.unlock();
    }
  }
}
```

Read-write locks allow shared state to be accessed by multiple readers or a single writer but never both. According to Goetz [Goetz 2006]

> *In practice, read-write locks can improve performance for frequently accessed read-mostly data structures on multiprocessor systems; under other conditions they perform slightly worse than exclusive locks due to their greater complexity.*

Profiling the application can determine the suitability of read-write locks.

2.3.7 Compliant Solution (`AtomicBoolean`)

This compliant solution declares `flag` an `AtomicBoolean` type.

```
import java.util.concurrent.atomic.AtomicBoolean;

final class Flag {
  private AtomicBoolean flag = new AtomicBoolean(true);

  public void toggle() {
    boolean temp;
    do {
      temp = flag.get();
    } while (!flag.compareAndSet(temp, !temp));
  }

  public AtomicBoolean getFlag() {
    return flag;
  }
}
```

The `flag` variable is updated using the `compareAndSet()` method of the `AtomicBoolean` class. All updates are visible to other threads.

2.3.8 Noncompliant Code Example (Addition of Primitives)

In this noncompliant code example, multiple threads can invoke the `setValues()` method to set the `a` and `b` fields. Because this class does not test for integer overflow, a user of the `Adder` class must ensure that the arguments to the `setValues()` method can be added without overflow. (For more information, see guideline "INT00-J. Perform explicit range checking to ensure integer operations do not overflow.[1]")

```
final class Adder {
  private int a;
  private int b;

  public int getSum() {
    return a + b;
  }
```

[1] This guideline is described at https://www.securecoding.cert.org/confluence/display/java/.

```
  public void setValues(int a, int b) {
    this.a = a;
    this.b = b;
  }
}
```

The `getSum()` method contains a race condition. For example, if `a` and `b` currently have the values 0 and `Integer.MAX_VALUE`, respectively, and one thread calls `getSum()` while another calls `setValues(Integer.MAX_VALUE, 0)`, the `getSum()` method might return 0 or `Integer.MAX_VALUE`, or it might overflow and wrap. Overflow will occur when the first thread reads `a` and `b` after the second thread has set the value of `a` to `Integer.MAX_VALUE`, but before it has set the value of `b` to 0.

Note that declaring the variables volatile does not resolve the issue because these compound operations involve reads and writes of multiple variables.

2.3.9 Noncompliant Code Example (Addition of Atomic Integers)

In this noncompliant code example, `a` and `b` are replaced with atomic integers.

```
final class Adder {
  private final AtomicInteger a = new AtomicInteger();
  private final AtomicInteger b = new AtomicInteger();

  public int getSum() {
    return a.get() + b.get();
  }

  public void setValues(int a, int b) {
    this.a.set(a);
    this.b.set(b);
  }
}
```

The simple replacement of the two `int` fields with atomic integers in this example does not eliminate the race condition because the compound operation `a.get() + b.get()` is still nonatomic.

2.3.10 Compliant Solution (Addition)

This compliant solution synchronizes the `setValues()` and `getSum()` methods to ensure atomicity.

```
final class Adder {
  private int a;
  private int b;

  public synchronized int getSum() {
    return a + b;
```

```
  }

  public synchronized void setValues(int a, int b) {
    this.a = a;
    this.b = b;
  }
}
```

Any operations within the synchronized methods are now atomic with respect to other synchronized methods that lock on that object's monitor (intrinsic lock). It is now possible, for example, to add overflow checking to the synchronized `getSum()` method without introducing the possibility of a race condition.

2.3.11 Risk Assessment

If operations on shared variables are non-atomic, unexpected results can be produced. For example, information can be disclosed inadvertently because one user can receive information about other users.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
VNA02-J	medium	probable	medium	P8	L2

2.3.12 References

[Bloch 2008] Item 66: Synchronize access to shared mutable data

[Goetz 2006] Section 2.3, "Locking"

[Gosling 2005] Chapter 17, "Threads and Locks"
Section 17.4.5, "Happens-Before Order"
Section 17.4.3, "Programs and Program Order"
Section 17.4.8, "Executions and Causality Requirements"

[Lea 2000a] Section 2.2.7, The Java Memory Model
Section 2.1.1.1, Objects and Locks

[MITRE 2010] CWE ID 667, "Insufficient Locking"
CWE ID 413, "Insufficient Resource Locking"
CWE ID 366, "Race Condition within a Thread"
CWE ID 567, "Unsynchronized Access to Shared Data"

[Sun 2009b] Class AtomicInteger

[Sun 2008a] Java Concurrency Tutorial

2.4 VNA03-J. Do not assume that a group of calls to independently atomic methods is atomic

A consistent locking policy guarantees that multiple threads cannot simultaneously access or modify shared data. If two or more operations need to be performed as a single atomic operation, a consistent locking policy must be implemented using either intrinsic synchronization or `java.util.concurrent` utilities. In the absence of such a policy, the code is susceptible to race conditions.

Given an invariant involving multiple objects, a programmer may incorrectly assume that individually atomic operations require no additional locking. Similarly, programmers may incorrectly assume that using a thread-safe `Collection` does not require explicit synchronization to preserve an invariant that involves the collection's elements. A thread-safe class can only guarantee atomicity of its individual methods. A grouping of calls to such methods requires additional synchronization.

Consider, for example, a scenario where the standard thread-safe API does not provide a single method to both find a particular person's record in a `Hashtable` and update the corresponding payroll information. In such cases, the two method invocations must be performed atomically.

Enumerations and iterators also require explicit synchronization on the collection object (client-side locking) or a private final lock object.

Compound operations on shared variables are also non-atomic. For more information, see guideline "VNA02-J. Ensure that compound operations on shared variables are atomic" on page 16.

Guideline "VNA04-J. Ensure that calls to chained methods are atomic" on page 29 describes a specialized case of this guideline.

2.4.1 Noncompliant Code Example (`AtomicReference`)

This noncompliant code example wraps `BigInteger` objects within thread-safe `AtomicReference` objects.

```
final class Adder {
  private final AtomicReference<BigInteger> first;
  private final AtomicReference<BigInteger> second;

  public Adder(BigInteger f, BigInteger s) {
    first  = new AtomicReference<BigInteger>(f);
    second = new AtomicReference<BigInteger>(s);
  }

  public void update(BigInteger f, BigInteger s) { // Unsafe
    first.set(f);
    second.set(s);
  }

  public BigInteger add() { // Unsafe
```

```
    return first.get().add(second.get());
  }
}
```

`AtomicReference` is an object reference that can be updated atomically. However, operations that combine more than one atomic reference are non-atomic. In this noncompliant code example, one thread may call `update()` while a second thread may call `add()`. This might cause the `add()` method to add the new value of `first` to the old value of `second`, yielding an erroneous result.

2.4.2 Compliant Solution (Method Synchronization)

This compliant solution declares the `update()` and `add()` methods synchronized to guarantee atomicity.

```
final class Adder {
  // ...

  public synchronized void update(BigInteger f, BigInteger s){
    first.set(f);
    second.set(s);
  }

  public synchronized BigInteger add() {
    return first.get().add(second.get());
  }
}
```

2.4.3 Noncompliant Code Example (`synchronizedList`)

This noncompliant code example uses a `java.util.ArrayList<E>` collection, which is not thread-safe. However, `Collections.synchronizedList` is used as a synchronization wrapper for `ArrayList`. An array, rather than an iterator, is used to iterate over `ArrayList` to avoid a `ConcurrentModificationException`.

```
final class IPHolder {
  private final List<InetAddress> ips =
    Collections.synchronizedList(new ArrayList<InetAddress>());

  public void addAndPrintIPAddresses(InetAddress address) {
    ips.add(address);
    InetAddress[] addressCopy = (InetAddress[]) ips.toArray(new InetAddress[0]);
    // Iterate through array addressCopy ...
  }
}
```

Individually, the `add()` and `toArray()` collection methods are atomic. However, when they are called in succession (for example, in the `addAndPrintIPAddresses()` method), there are no guarantees that the combined operation is atomic. A race condition exists in the

`addAndPrintIPAddresses()` method that allows one thread to add to the list and a second thread to race in and modify the list before the first thread completes. Consequently, the `addressCopy` array may contain more IP addresses than expected.

2.4.4 Compliant Solution (Synchronized Block)

The race condition can be eliminated by synchronizing on the underlying list's lock. This compliant solution encapsulates all references to the array list within synchronized blocks.

```
final class IPHolder {
  private final List<InetAddress> ips =
    Collections.synchronizedList(new ArrayList<InetAddress>());

  public void addAndPrintIPAddresses(InetAddress address) {
    synchronized (ips) {
      ips.add(address);
      InetAddress[] addressCopy = (InetAddress[]) ips.toArray(new InetAddress[0]);
      // Iterate through array addressCopy ...
    }
  }
}
```

This technique is also called client-side locking [Goetz 2006] because the class holds a lock on an object that might be accessible to other classes. Client-side locking is not always an appropriate strategy; see guideline "LCK11-J. Avoid client-side locking when using classes that do not commit to their locking strategy" on page 86 for more information.

This code does not violate guideline "LCK04-J. Do not synchronize on a collection view if the backing collection is accessible" on page 57 because, while it does synchronize on a collection view (the `synchronizedList`), the backing collection is inaccessible and therefore cannot be modified by any code.

2.4.5 Noncompliant Code Example (`synchronizedMap`)

This noncompliant code example defines the `KeyedCounter` class that is not thread-safe. Although the `HashMap` is wrapped in a `synchronizedMap`, the overall increment operation is non-atomic [Lee 2009].

```
final class KeyedCounter {
  private final Map<String, Integer> map =
    Collections.synchronizedMap(new HashMap<String, Integer>());

  public void increment(String key) {
    Integer old = map.get(key);
    int oldValue = (old == null) ? 0 : old.intValue();
    if (oldValue == Integer.MAX_VALUE) {
      throw new ArithmeticException("Out of range");
    }
    map.put(key, oldValue + 1);
```

```
  }

  public Integer getCount(String key) {
    return map.get(key);
  }
}
```

2.4.6 Compliant Solution (Synchronization)

To ensure atomicity, this compliant solution uses an internal private lock object to synchronize the statements of the `increment()` and `getCount()` methods.

```
final class KeyedCounter {
  private final Map<String, Integer> map = new HashMap<String, Integer>();
  private final Object lock = new Object();

  public void increment(String key) {
    synchronized (lock) {
      Integer old = map.get(key);
      int oldValue = (old == null) ? 0 : old.intValue();
      if (oldValue == Integer.MAX_VALUE) {
        throw new ArithmeticException("Out of range");
      }
      map.put(key, oldValue + 1);
    }
  }

  public Integer getCount(String key) {
    synchronized (lock) {
      return map.get(key);
    }
  }
}
```

This compliant solution does not use `Collections.synchronizedMap()` because locking on the unsynchronized map provides sufficient thread-safety for this application. Guideline “LCK04-J. Do not synchronize on a collection view if the backing collection is accessible” on page 57 provides more information about synchronizing on `synchronizedMap` objects.

2.4.7 Compliant Solution (`ConcurrentHashMap`)

The previous compliant solution is safe for multithreaded use, but it does not scale well because of excessive synchronization, which can lead to contention and deadlock.

The `ConcurrentHashMap` class used in this compliant solution provides several utility methods for performing atomic operations and is often a good choice for algorithms that must scale [Lee 2009].

```
final class KeyedCounter {
  private final ConcurrentMap<String, AtomicInteger> map =
```

```
    new ConcurrentHashMap<String, AtomicInteger>();

  public void increment(String key) {
    AtomicInteger value = new AtomicInteger();
    AtomicInteger old = map.putIfAbsent(key, value);

    if (old != null) {
      value = old;
    }

    if (value.get() == Integer.MAX_VALUE) {
      throw new ArithmeticException("Out of range");
    }

    value.incrementAndGet(); // Increment the value atomically
  }

  public Integer getCount(String key) {
    AtomicInteger value = map.get(key);
    return (value == null) ? null : value.get();
  }

  // Other accessors ...
}
```

According to Section 5.2.1., "ConcurrentHashMap" of the work of Goetz and colleagues [Goetz 2006]

> *`ConcurrentHashMap`, along with the other concurrent collections, further improve on the synchronized collection classes by providing iterators that do not throw `ConcurrentModificationException`, as a result eliminating the need to lock the collection during iteration. The iterators returned by `ConcurrentHashMap` are weakly consistent instead of fail-fast. A weakly consistent iterator can tolerate concurrent modification, traverses elements as they existed when the iterator was constructed, and may (but is not guaranteed to) reflect modifications to the collection after the construction of the iterator.*

Note that methods such as `ConcurrentHashMap.size()` and `ConcurrentHashMap.isEmpty()` are allowed to return an approximate result for performance reasons. Code should not rely on these return values for deriving exact results.

2.4.8 Risk Assessment

Failing to ensure the atomicity of two or more operations that need to be performed as a single atomic operation can result in race conditions in multithreaded applications.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
VNA03- J	low	probable	medium	**P4**	**L3**

2.4.9 References

[Goetz 2006] Section 4.4.1, “Client-side Locking”
Section 5.2.1, “ConcurrentHashMap”

[Lee 2009] “Map & Compound Operation”

[Oaks 2004] Section 8.2, “Synchronization and Collection Classes”

[Sun 2009c]

2.5 VNA04-J. Ensure that calls to chained methods are atomic

Method chaining is a convenience mechanism that allows multiple method invocations on the same object to occur in a single statement. A method-chaining implementation consists of a series of methods that return the `this` reference. This implementation allows a caller to invoke methods in a chain by performing the next method invocation on the return value of the previous method in the chain.

While the methods used in method chaining can be atomic, the chain they comprise is inherently non-atomic. Consequently, methods that are involved in method chaining should not be invoked concurrently unless the caller provides sufficient locking as illustrated in guideline “VNA03-J. Do not assume that a group of calls to independently atomic methods is atomic” on page 23.

2.5.1 Noncompliant Code Example

Method chaining is a useful design pattern for building an object and setting its optional fields. A class that supports method chaining provides several setter methods that each return the `this` reference. However, if accessed concurrently, a thread may observe shared fields to contain inconsistent values. This noncompliant code example shows the JavaBeans pattern, which is not thread-safe.

```
final class USCurrency {
  // Change requested, denomination (optional fields)
  private int quarters = 0;
  private int dimes = 0;
  private int nickels = 0;
  private int pennies = 0;

  public USCurrency() {}

  // Setter methods
  public USCurrency setQuarters(int quantity) {
    quarters = quantity;
    return this;
  }
  public USCurrency setDimes(int quantity) {
    dimes = quantity;
    return this;
  }
  public USCurrency setNickels(int quantity) {
    nickels = quantity;
    return this;
  }
  public USCurrency setPennies(int quantity) {
    pennies = quantity;
    return this;
  }
}
```

```
// Client code:
private final USCurrency currency = new USCurrency();
// ...

new Thread(new Runnable() {
  @Override public void run() {
    currency.setQuarters(1).setDimes(1);
  }
}).start();

new Thread(new Runnable() {
  @Override public void run() {
    currency.setQuarters(2).setDimes(2);
  }
}).start();
```

The JavaBeans pattern uses a no-argument constructor and a series of parallel setter methods to build an object. This pattern is not thread-safe and can lead to inconsistent object state if the object is modified concurrently. In this noncompliant code example, the client constructs a `USCurrency` object and starts two threads that use method chaining to set the optional values of the `USCurrency` object. This example code might result in the `USCurrency` instance being left in an inconsistent state, for example, with two quarters and one dime, or one quarter and two dimes.

2.5.2 Compliant Solution

This compliant solution uses a variant of the Builder pattern [Gamma 1995] suggested by Bloch [Bloch 2008] to ensure the thread-safety and atomicity of object creation.

```
final class USCurrency {
  private final int quarters;
  private final int dimes;
  private final int nickels;
  private final int pennies;

  public USCurrency(Builder builder) {
    this.quarters = builder.quarters;
    this.dimes = builder.dimes;
    this.nickels = builder.nickels;
    this.pennies = builder.pennies;
  }

  // Static class member
  public static class Builder {
    private int quarters = 0;
    private int dimes = 0;
    private int nickels = 0;
    private int pennies = 0;
```

```
    public static Builder newInstance() {
      return new Builder();
    }

    private Builder() {}

    // Setter methods
    public Builder setQuarters(int quantity) {
      this.quarters = quantity;
      return this;
    }
    public Builder setDimes(int quantity) {
      this.dimes = quantity;
      return this;
    }
    public Builder setNickels(int quantity) {
      this.nickels = quantity;
      return this;
    }
    public Builder setPennies(int quantity) {
      this.pennies = quantity;
      return this;
    }

    public USCurrency build() {
      return new USCurrency(this);
    }
  }
}

  // Client code:
private volatile USCurrency currency;
// ...

new Thread(new Runnable() {
  @Override public void run() {
    currency = USCurrency.Builder.newInstance().setQuarters(1).setDimes(1).build();
  }
}).start();

new Thread(new Runnable() {
  @Override public void run() {
    currency = USCurrency.Builder.newInstance().setQuarters(2).setDimes(2).build();
  }
}).start();
```

The `Builder.newInstance()` factory method is called with any required arguments to obtain a `Builder` instance. The optional parameters are set using the setter methods of the builder. The

object construction concludes with the invocation of the `build()` method. This pattern makes the `USCurrency` class immutable and, consequently, thread-safe.

Note that the `currency` field cannot be declared final because it is assigned a new immutable object. It is, however, declared volatile in compliance with guideline "VNA01-J. Ensure visibility of shared references to immutable objects" on page 13.

If input needs to be validated, ensure that the values are defensively copied prior to validation (see guideline "FIO00-J. Defensively copy mutable inputs and mutable internal components[2]" for more information). The `builder` class does not violate guideline "SCP03-J. Do not expose sensitive private members of the outer class from within a nested class[2]" because it maintains a copy of the variables defined in the scope of the containing class. The private members within the nested class take precedence, and as a result, do not break encapsulation.

2.5.3 Risk Assessment

Using method chaining in multithreaded environments without performing external locking can lead to nondeterministic behavior.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
VNA04- J	low	probable	medium	**P4**	**L3**

2.5.4 References

[Bloch 2008] Item 2: "Consider a builder when faced with many constructor parameters"

[Sun 2009b]

[2] This guideline is described at https://www.securecoding.cert.org/confluence/display/java/.

2.6 VNA05-J. Ensure atomicity when reading and writing 64-bit values

The *Java Language Specification* allows 64-bit `long` and `double` values to be treated as two 32-bit values. For example, a 64-bit write operation may be performed as two separate, 32-bit operations.

According to the *Java Language Specification*, Section 17.7, "Non-atomic Treatment of `double` and `long`" [Gosling 2005]

> *... this behavior is implementation specific; Java virtual machines are free to perform writes to `long` and `double` values atomically or in two parts. For the purposes of the Java programming language memory model, a single write to a non-volatile `long` or `double` value is treated as two separate writes: one to each 32-bit half. This can result in a situation where a thread sees the first 32 bits of a 64 bit value from one write, and the second 32 bits from another write.*

This behavior can result in indeterminate values being read in code that is required to be thread-safe.

2.6.1 Noncompliant Code Example

In this noncompliant code example, if one thread repeatedly calls the `assignValue()` method and another thread repeatedly calls the `printLong()` method, the `printLong()` method could occasionally print a value of `i` that is neither zero nor the value of the `j` argument.

```
class LongContainer {
  private long i = 0;

  void assignValue(long j) {
    i = j;
  }

  void printLong() {
    System.out.println("i = " + i);
  }
}
```

A similar problem may occur if `i` is declared double.

2.6.2 Compliant Solution (Volatile)

This compliant solution declares `i` volatile. Writes and reads of `long` and `double` volatile values are always atomic.

```
class LongContainer {
  private volatile long i = 0;

  void assignValue(long j) {
    i = j;
  }
```

```
  void printLong() {
    System.out.println("i = " + i);
  }
}
```

It is important to ensure that the argument to the `assignValue()` method is obtained from a volatile variable or as a result of explicitly passing an integer value. Otherwise, a read of the variable argument can, itself, expose a vulnerability.

Semantics of `volatile` do not guarantee the atomicity of compound operations that involve read-modify-write sequences such as incrementing a value. See guideline "VNA02-J. Ensure that compound operations on shared variables are atomic" on page 16 for more information.

2.6.3 Exceptions

VNA05-EX1: If all reads and writes of 64-bit `long` and `double` values occur within a synchronized region, the atomicity of the read/write is guaranteed. That guarantee requires that no unsynchronized methods in the class expose the value and that the value is inaccessible (directly or indirectly) from other code. (For more information, see guideline "VNA02-J. Ensure that compound operations on shared variables are atomic" on page 16.)

VNA05-EX2: This guideline can be ignored for systems that guarantee that 64-bit, `long` and `double` values are read and written as atomic operations.

2.6.4 Risk Assessment

Failure to ensure the atomicity of operations involving 64-bit values in multithreaded applications can result in reading and writing indeterminate values. Many JVMs read and write 64-bit values atomically, even though the specification does not require them to.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
VNA05- J	low	unlikely	medium	P2	L3

2.6.5 References

[Goetz 2004c]

[Goetz 2006] Section 3.1.2, "Non-Atomic 64-Bit Operations"

[Gosling 2005] Section 17.7, "Non-Atomic Treatment of double and long"

[MITRE 2010] CWE ID 667, "Insufficient Locking"

2.7 VNA06-J. Do not assume that declaring an object reference volatile guarantees visibility of its members

According to the *Java Language Specification*, Section 8.3.1.4, "`volatile` Fields" [Gosling 2005]

> *A field may be declared `volatile`, in which case the Java memory model (§17) ensures that all threads see a consistent value for the variable.*

Notably, this applies only to primitive fields and immutable member objects. The visibility guarantee does not extend to non-thread-safe mutable objects, even if their references are declared volatile. A thread may not observe a recent write from another thread to a member field of such an object. Declaring an object volatile to ensure the visibility of its state does not work without the use of synchronization, unless the object is immutable. If the object is mutable and not thread-safe, other threads might see a partially constructed object or an object in a (temporarily) inconsistent state [Goetz 2006c].

Technically, the object does not have to be strictly immutable to be used safely. If it can be determined that a member object is thread-safe by design, the field that holds its reference may be declared volatile. However, this approach to declaring elements volatile decreases maintainability and should be avoided.

2.7.1 Noncompliant Code Example (Arrays)

This noncompliant code example shows an array object that is declared volatile.

```
final class Foo {
  private volatile int[] arr = new int[20];

  public int getFirst() {
    return arr[0];
  }

  public void setFirst(int n) {
    arr[0] = n;
  }

  // ...
}
```

Values assigned to an array element by one thread, for example, by calling `setFirst()`, might not be visible to another thread calling `getFirst()` because the `volatile` keyword only makes the array reference visible and does not affect the actual data contained within the array.

The problem occurs because there is no happens-before relationship between the thread that calls `setFirst()` and the thread that calls `getFirst()`. A happens-before relationship exists between a thread that writes to a volatile variable and a thread that subsequently reads it. However, this code is neither writing to nor reading from a volatile variable.

2.7.2 Compliant Solution (`AtomicIntegerArray`)

To ensure that the writes to array elements are atomic and the resulting values are visible to other threads, this compliant solution uses the `AtomicIntegerArray` class defined in `java.util.concurrent.atomic`.

```
final class Foo {
  private final AtomicIntegerArray atomicArray = new AtomicIntegerArray(20);

  public int getFirst() {
    return atomicArray.get(0);
  }

  public void setFirst(int n) {
    atomicArray.set(0, 10);
  }

  // ...
}
```

`AtomicIntegerArray` guarantees a happens-before relationship between a thread that calls `atomicArray.set()` and a thread that subsequently calls `atomicArray.get()`.

2.7.3 Compliant Solution (Synchronization)

To ensure visibility, accessor methods may synchronize access, while performing operations on non-volatile elements of an array that is declared volatile. Note that the code is thread-safe, even though the array reference is non-volatile.

```
final class Foo {
  private int[] arr = new int[20];

  public synchronized int getFirst() {
    return arr[0];
  }

  public synchronized void setFirst(int n) {
    arr[0] = n;
  }
}
```

Synchronization establishes a happens-before relationship between the thread that calls `setFirst()` and the thread that subsequently calls `getFirst()`, guaranteeing visibility.

2.7.4 Noncompliant Code Example (Mutable Object)

This noncompliant code example declares the `Properties` instance field volatile. The instance of the `Properties` object can be mutated using the `put()` method, and that makes the `properties` field mutable.

```
final class Foo {
  private volatile Properties properties;

  public Foo() {
    properties = new Properties();
    // Load some useful values into properties
  }

  public String get(String s) {
    return properties.getProperty(s);
  }

  public void put(String key, String value) {
    // Validate the values before inserting
    if (!value.matches("[\\w]*")) {
        throw new IllegalArgumentException();
    }
    properties.setProperty(key, value);
  }
```

Interleaved calls to `get()` and `put()` may result in internally inconsistent values being retrieved from the `Properties` object because the operations within `put()` modify its state. Declaring the object volatile does not eliminate this data race.

There is no time-of-check-to-time-of-use (TOCTOU) vulnerability in `put()`, despite the presence of the validation logic because the validation is performed on the immutable `value` argument and not the shared `Properties` instance.

2.7.5 Noncompliant Code Example (Volatile-Read, Synchronized-Write)

This noncompliant code example attempts to use the volatile-read, synchronized-write technique described by Goetz [Goetz 2006c]. The `properties` field is declared volatile to synchronize its reads and writes. The `put()` method is also synchronized to ensure that its statements are executed atomically.

```
final class Foo {
  private volatile Properties properties;

  public Foo() {
    properties = new Properties();
    // Load some useful values into properties
  }

  public String get(String s) {
    return properties.getProperty(s);
  }

  public synchronized void put(String key, String value) {
```

```
    // Validate the values before inserting
    if (!value.matches("[\\w]*")) {
        throw new IllegalArgumentException();
    }
    properties.setProperty(key, value);
  }
}
```

The volatile-read, synchronized-write technique uses synchronization to preserve the atomicity of compound operations, such as increment, and provides faster access times for atomic reads. However, it does not work with mutable objects because the visibility of `volatile` object references does not extend to object members. Consequently, there is no happens-before relationship between the write and a subsequent read of the property.

This technique is also discussed in guideline “VNA02-J. Ensure that compound operations on shared variables are atomic” on page 16.

2.7.6 Compliant Solution (Synchronization)

This compliant solution uses method synchronization to guarantee visibility.

```
final class Foo {
  private final Properties properties;

  public Foo() {
    properties = new Properties();
    // Load some useful values into properties
  }

  public synchronized String get(String s) {
    return properties.getProperty(s);
  }

  public synchronized void put(String key, String value) {
    // Validate the values before inserting
    if (!value.matches("[\\w]*")) {
      throw new IllegalArgumentException();
    }
    properties.setProperty(key, value);
  }
}
```

The `properties` field does not need to be volatile because the methods are synchronized. The field is declared final so that its reference is not published when it is in a partially initialized state (see guideline “TSM03-J. Do not publish partially initialized objects” on page 162 for more information).

2.7.7 Noncompliant Code Example (Mutable Sub-Object)

In this noncompliant code example, the volatile `format` field is used to store a reference to a mutable object, `java.text.DateFormat`.

```
final class DateHandler {
  private static volatile DateFormat format =
    DateFormat.getDateInstance(DateFormat.MEDIUM);

  public static Date parse(String str) throws ParseException {
    return format.parse(str);
  }
}
```

Because `DateFormat` is not thread-safe [Sun 2009c], the `parse()` method might return a value for `Date` that does not correspond to the `str` argument.

2.7.8 Compliant Solution (Instance Per Call/Defensive Copying)

This compliant solution creates and returns a new `DateFormat` instance for every invocation of the `parse()` method [Sun 2009c].

```
final class DateHandler {
  public static Date parse(String str) throws ParseException {
    return DateFormat.getDateInstance(DateFormat.MEDIUM).parse(str);
  }
}
```

This solution does not violate guideline "OBJ11-J. Defensively copy private mutable class members before returning their references"[3] because the class no longer contains internal mutable state.

2.7.9 Compliant Solution (Synchronization)

This compliant solution synchronizes statements within the `parse()` method, making `DateHandler` thread-safe [Sun 2009c].

```
final class DateHandler {
  private static DateFormat format=
    DateFormat.getDateInstance(DateFormat.MEDIUM);

  public static Date parse(String str) throws ParseException {
    synchronized (format) {
      return format.parse(str);
    }
  }
}
```

[3] This guideline is described at https://www.securecoding.cert.org/confluence/display/java/.

2.7.10 Compliant Solution (`ThreadLocal` Storage)

This compliant solution uses a `ThreadLocal` object to create a separate `DateFormat` instance per thread.

```
final class DateHandler {
  private static final ThreadLocal<DateFormat> format =
    new ThreadLocal<DateFormat>() {
    @Override protected DateFormat initialValue() {
      return DateFormat.getDateInstance(DateFormat.MEDIUM);
    }
  };
  // ...
}
```

2.7.11 Risk Assessment

Incorrectly assuming that declaring a field volatile guarantees that the visibility of a referenced object's members can cause threads to observe stale values.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
VNA06-J	medium	probable	medium	P8	L2

2.7.12 References

[Goetz 2006c] Pattern #2: "one-time safe publication"

[Gosling 2005]

[Miller 2009] Mutable Statics

[Sun 2009c] Class `java.text.DateFormat`

3 Lock (LCK) Guidelines

3.1 LCK00-J. Use private final lock objects to synchronize classes that may interact with untrusted code

The `synchronized` keyword is used to acquire a mutual-exclusion lock so that no other thread can acquire the lock while it is being held by the executing thread. There are two ways to synchronize access to shared mutable variables: method synchronization and block synchronization.

A method declared as synchronized always uses the object's monitor (intrinsic lock), as does code that synchronizes on the `this` reference using a synchronized block. Poorly synchronized code is prone to contention and deadlock. An attacker can manipulate the system to trigger these conditions and cause a denial of service by obtaining and indefinitely holding the intrinsic lock of an accessible class.

This vulnerability can be prevented using a `java.lang.Object` declared `private` and `final` within the class. The object must be used explicitly for locking purposes in synchronized blocks within the class's methods. This intrinsic lock is associated with the instance of the private object and not the class. Consequently, there is no lock contention between this class's methods and the methods of a hostile class. Bloch refers to this technique as the "private lock object" idiom [Bloch 2001].

Static state has the same potential problem. If a static method is declared synchronized, the intrinsic lock of the class object is acquired before any statements in its body are executed, and the lock is released when the method completes. Any untrusted code that can access an object of the class, or a subclass, can use the `getClass()` method to gain access to the class lock. Static data can be protected by locking on a private static final `Object`. Reducing the accessibility of the class to package-private adds further protection against untrusted callers.

This idiom is also suitable for classes designed for inheritance. If a superclass thread requests a lock on the object's monitor, a subclass thread can interfere with its operation. For example, a subclass may use the superclass object's intrinsic lock for performing unrelated operations, causing significant lock contention and deadlock. Separating the locking strategy of the superclass from that of the subclass ensures that they do not share a common lock. It also permits fine-grained locking because multiple lock objects can be used for unrelated operations, increasing the overall responsiveness of the application.

An object should use a private final lock object rather than its own intrinsic lock unless the class can guarantee that untrusted code cannot

- subclass the class or its superclass (trusted code is allowed to subclass the class)
- create an object of the class, its superclass, or subclass
- access or acquire an object instance of the class, its superclass, or subclass

If a class uses a private final lock to synchronize shared data, subclasses must also use a private final lock. However, if a class uses intrinsic synchronization over the class object without documenting its locking policy, subclasses may not use intrinsic synchronization over their own class

object, unless they explicitly document their locking policy. If the superclass documents its policy by stating that client-side locking is supported, the subclasses have the option of choosing between intrinsic locking over the class object and a private lock. Regardless of which is chosen, subclasses must document their locking policy. See guideline "TSM00-J. Do not override thread-safe methods with methods that are not thread-safe" on page 145 for related information.

If all of these restrictions are not met, the object's intrinsic lock is not trustworthy. If they are met, the object gains no significant security from using a private final lock object and may synchronize using its own intrinsic lock. However, it is still best to use block synchronization with a private final lock object instead of method synchronization when the method contains non-atomic operations that either do not require any synchronization or can use a more fine-grained locking scheme involving multiple private final lock objects. Non-atomic operations can be decoupled from those that require synchronization and executed outside the synchronized block. For this reason and maintainability reasons, block synchronization using a private final lock object is generally recommended.

3.1.1 Noncompliant Code Example (Method Synchronization)

This noncompliant code example exposes instances of the `SomeObject` class to untrusted code.

```
public class SomeObject {
  public synchronized void changeValue() { // Locks on the object's monitor
    // ...
  }
}

// Untrusted code
SomeObject someObject = new SomeObject();
synchronized (someObject) {
  while (true) {
    Thread.sleep(Integer.MAX_VALUE); // Indefinitely delay someObject
  }
}
```

The untrusted code attempts to acquire a lock on the object's monitor and, upon succeeding, introduces an indefinite delay that prevents the `synchronized changeValue()` method from acquiring the same lock. Note that in the untrusted code, the attacker intentionally violates guideline "LCK09-J. Do not perform operations that may block while holding a lock" on page 77.

3.1.2 Noncompliant Code Example (Public Non-Final Lock Object)

This noncompliant code example locks on a public non-final object in an attempt to use a lock other than `SomeObject`'s intrinsic lock.

```
public class SomeObject {
  public Object lock = new Object();

  public void changeValue() {
    synchronized (lock) {
```

```
      // ...
    }
  }
}
```

However, it is possible for untrusted code to change the value of the lock object and disrupt proper synchronization.

3.1.3 Noncompliant Code Example (Publicly Accessible Non-Final Lock Object)

This noncompliant code example synchronizes on a private but non-final field.

```
public class SomeObject {
  private volatile Object lock = new Object();

  public void changeValue() {
    synchronized (lock) {
      // ...
    }
  }

  public void setLock(Object lockValue) {
    lock = lockValue;
  }
}
```

Any thread can modify the field's value to refer to a different object in the presence of an accessor such as `setLock()`. That modification might cause two threads that intend to lock on the same object to lock on different objects, thereby enabling them to execute the two critical sections in an unsafe manner. For example, if one thread is in its critical section and the lock is changed, a second thread will lock on the new object instead of the old one.

A class that does not provide any accessible methods to change the lock is secure against untrusted manipulation. However, it is susceptible to inadvertent modification by the programmer. For maintainability reasons, eliminating the accessor method (which is presumably needed for other reasons) is not the preferred solution.

3.1.4 Noncompliant Code Example (Public Final Lock Object)

This noncompliant code example uses a public final lock object.

```
public class SomeObject {
  public final Object lock = new Object();

  public void changeValue() {
    synchronized (lock) {
      // ...
    }
  }
}
```

```
// Untrusted code
SomeObject someObject = new SomeObject();
someObject.lock.wait()
```

Untrusted code that has the ability to create an instance of the class or has access to an already created instance can invoke the `wait()` method on the publicly accessible `lock`, causing the lock in the `changeValue()` method to be released immediately. Furthermore, if the method invokes `lock.wait()` from its body and does not test a condition predicate, it will be vulnerable to malicious notifications. (See guideline "THI03-J. Always invoke wait() and await() methods inside a loop" on page 101 for more information.)

3.1.5 Compliant Solution (Private Final Lock Object)

Thread-safe public classes that may interact with untrusted code must use a private final lock object. Existing classes that use intrinsic synchronization must be refactored to use block synchronization on such an object. In this compliant solution, calling `changeValue()` obtains a lock on a private final `Object` instance that is inaccessible from callers outside the class's scope.

```
public class SomeObject {
  private final Object lock = new Object(); // private final lock object

  public void changeValue() {
    synchronized (lock) { // Locks on the private Object
      // ...
    }
  }
}
```

A private final lock object can only be used with block synchronization. Block synchronization is preferred over method synchronization, because operations that do not require synchronization can be moved outside the synchronized region, reducing lock contention and blocking. Note that there is no need to declare `lock` volatile because of the strong visibility semantics of final fields. Instead of using setter methods to change the lock, declare and use multiple, private final lock objects to satisfy the granularity requirements.

3.1.6 Noncompliant Code Example (Static)

This noncompliant code example exposes the class object of `SomeObject` to untrusted code.

```
public class SomeObject {
  // changeValue locks on the class object's monitor
  public static synchronized void changeValue() {
    // ...
  }
}

// Untrusted code
synchronized (SomeObject.class) {
```

```
  while (true) {
    Thread.sleep(Integer.MAX_VALUE); // Indefinitely delay someObject
  }
}
```

The untrusted code attempts to acquire a lock on the class object's monitor and, upon succeeding, introduces an indefinite delay that prevents the synchronized `changeValue()` method from acquiring the same lock.

A compliant solution must comply with guideline "LCK05-J. Synchronize access to static fields that may be modified by untrusted code" on page 59. However, in the untrusted code, the attacker intentionally violates guideline "LCK09-J. Do not perform operations that may block while holding a lock" on page 77.

3.1.7 Compliant Solution (Static)

Thread-safe public classes that may interact with untrusted code and use intrinsic synchronization over the class object must be refactored to use a static private final lock object and block synchronization.

```
public class SomeObject {
  private static final Object lock = new Object(); // private final lock object

  public static void changeValue() {
    synchronized (lock) { // Locks on the private Object
      // ...
    }
  }
}
```

In this compliant solution, `changeValue()` obtains a lock on a static private `Object` that is inaccessible to the caller.

3.1.8 Exceptions

LCK00-EX1: A class may violate this guideline, if all the following conditions are met:

- It sufficiently documents that callers must not pass objects of this class to untrusted code.
- The class does not invoke methods on objects of any untrusted classes that violate this guideline directly or indirectly.
- The synchronization policy of the class is documented properly.

A client may use a class that violates this guideline, if all the following conditions are met:

- The class does not pass objects of this class to untrusted code.
- The class does not use any untrusted classes that violate this guideline directly or indirectly.

LCK00-EX2: If a superclass of the class documents that it supports client-side locking and synchronizes on its class object, the class can support client-side locking in the same way and document this policy.

LCK00-EX3: A package-private class may violate this guideline because its accessibility protects against untrusted callers. However, this condition should be documented explicitly so that trusted code within the same package does not reuse or change the lock object inadvertently.

3.1.9 Risk Assessment

Exposing the class object to untrusted code can result in a denial of service.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
LCK00-J	low	probable	medium	P4	L3

3.1.10 References

[Bloch 2001] Item 52: “Document Thread Safety”

3.2 LCK01-J. Do not synchronize on objects that may be reused

Misuse of synchronization primitives is a common source of concurrency issues. Synchronizing on objects that may be reused can result in deadlock and nondeterministic behavior.

3.2.1 Noncompliant Code Example (Boolean Lock Object)

This noncompliant code example synchronizes on a `Boolean` lock object.

```
private final Boolean initialized = Boolean.FALSE;

public void doSomething() {
  synchronized (initialized) {
    // ...
  }
}
```

The `Boolean` type is unsuitable for locking purposes because it allows only two values: true and false. Boolean literals containing the same value share unique instances of the `Boolean` class in the JVM. In this example, `initialized` references the instance corresponding to the value false. If any other code inadvertently synchronizes on a Boolean literal with the value false, the lock instance is reused and the system can become unresponsiveness or deadlocked.

3.2.2 Noncompliant Code Example (Boxed Primitive)

This noncompliant code example locks on a boxed `Integer` object.

```
int lock = 0;
private final Integer Lock = lock; // Boxed primitive Lock is shared

public void doSomething() {
  synchronized (Lock) {
    // ...
  }
}
```

Boxed types may use the same instance for a range of integer values and consequently suffer from the same problem as `Boolean` constants. If the value of the primitive can be represented as a byte, the wrapper object is reused. Note that the use of the boxed `Integer` wrapper object is insecure; instances of the `Integer` object constructed using the `new` operator (`new Integer(value)`) are unique and not reused. In general, holding a lock on any data type that contains a boxed value is insecure.

3.2.3 Compliant Solution (Integer)

This compliant solution recommends locking on a non-boxed `Integer`. The `doSomething()` method synchronizes using the intrinsic lock of the `Integer` instance, `Lock`.

```
int lock = 0;
private final Integer Lock = new Integer(lock);

public void doSomething() {
  synchronized (Lock) {
    // ...
  }
}
```

When explicitly constructed, an `Integer` object has a unique reference and its own intrinsic lock that is not shared with other `Integer` objects or boxed integers having the same value. While this is an acceptable solution, it can cause maintenance problems because developers can incorrectly assume that boxed integers are appropriate lock objects. A more appropriate solution is to synchronize on a private final lock object as described in the compliant solution in Section 3.2.7.

3.2.4 Noncompliant Code Example (Interned String Object)

This noncompliant code example locks on an interned `String` object.

```
private final String lock = new String("LOCK").intern();

public void doSomething() {
  synchronized (lock) {
    // ...
  }
}
```

According to the Java API class `java.lang.String` documentation [Sun 2009c]

> *When the `intern()` method is invoked, if the pool already contains a string equal to this `String` object as determined by the `equals(Object)` method, then the string from the pool is returned. Otherwise, this `String` object is added to the pool and a reference to this `String` object is returned.*

Consequently, an interned `String` object behaves like a global variable in the JVM. As demonstrated in this noncompliant code example, even if every instance of an object maintains its own `lock` field, the field references a common `String` constant. Locking on `String` constants has the same problem as locking on `Boolean` constants.

Additionally, hostile code from any other package can exploit this vulnerability, if the class is accessible. (For more information, see guideline “LCK00-J. Use private final lock objects to synchronize classes that may interact with untrusted code” on page 41.)

3.2.5 Noncompliant Code Example (`String` Literal)

This noncompliant code example locks on a final `String` literal.

```
// This bug was found in jetty-6.1.3 BoundedThreadPool
private final String lock = "LOCK";

// ...
  synchronized (lock) {
    // ...
  }
// ...
```

A `String` literal is a constant and interned. Consequently, it suffers from the same pitfalls as the preceding noncompliant code example.

3.2.6 Compliant Solution (`String` Instance)

This compliant solution locks on a `String` instance that is not interned.

```
private final String lock = new String("LOCK");

public void doSomething() {
  synchronized (lock) {
    // ...
  }
}
```

A `String` instance differs from a `String` literal. The instance has a unique reference and its own intrinsic lock that is not shared by other `String` object instances or literals. A better approach is to synchronize on a private final lock object as shown in the following compliant solution.

3.2.7 Compliant Solution (Private Final Lock Object)

This compliant solution synchronizes on a private final lock object. This is one of the few cases where a `java.lang.Object` instance is useful.

```
private final Object lock = new Object();

public void doSomething() {
  synchronized (lock) {
    // ...
  }
}
```

For more information on using an `Object` as a lock, see guideline “LCK00-J. Use private final lock objects to synchronize classes that may interact with untrusted code” on page 41.

3.2.8 Risk Assessment

A significant number of concurrency vulnerabilities arise from locking on the wrong kind of object. It is important to consider the properties of the lock object rather than indiscreetly scavenging for objects to synchronize on.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
LCK01- J	medium	probable	medium	P8	L2

3.2.9 References

[Findbugs 2008]

[Miller 2009] Locking

[Pugh 2008] “Synchronization”

[Sun 2009c] Class String, Collections

[Sun 2008a] Wrapper Implementations

3.3 LCK02-J. Do not synchronize on the class object returned by getClass()

Synchronizing on the return value of the `Object.getClass()` method can lead to unexpected behavior. Whenever the implementing class is subclassed, the subclass locks on the subclass's type, which is a completely different `Class` object.

Section 4.3.2, "The Class Object" of the *Java Language Specification* describes how method synchronization works [Gosling 2005]:

> *A class method that is declared `synchronized` synchronizes on the lock associated with the `Class` object of the class.*

This does not mean that a subclass using `getClass()` can only synchronize on the `Class` object of the base class. In fact, it will lock on its own `Class` object, which may or may not be what the programmer intended. The intent should be clearly documented or annotated. Note that if a subclass does not override an accessible noncompliant superclass's method, it inherits the method, which may lead to the false conclusion that the superclass's intrinsic lock is available in the subclass.

When synchronizing on a class literal, the corresponding lock object should not be accessible to untrusted code. If the class is package-private, callers from other packages may not access the class object, ensuring its trustworthiness as an intrinsic lock object. For more information, see guideline "LCK00-J. Use private final lock objects to synchronize classes that may interact with untrusted code" on page 41.

3.3.1 Noncompliant Code Example (`getClass()` Lock Object)

In this noncompliant code example, the `parse()` method of the `Base` class parses a date and synchronizes on the class object returned by `getClass()`. The `Derived` class also inherits the `parse()` method. However, this inherited method synchronizes on `Derived`'s class object because of the particular return value of `getClass()`.

The `Derived` class also adds a `doSomethingAndParse()`method that locks on the class object of the `Base` class because the developer misconstrued that the `parse()` method in `Base` always obtains a lock on the `Base` class object, and `doSomethingAndParse()` must follow the same locking policy. Consequently, the `Derived` class has two different locking strategies and is not thread-safe.

```
class Base {
  static DateFormat format =
    DateFormat.getDateInstance(DateFormat.MEDIUM);

  public Date parse(String str) throws ParseException {
    synchronized (getClass()) {
      return format.parse(str);
    }
  }
}
```

```
class Derived extends Base {
  public Date doSomethingAndParse(String str) throws ParseException {
    synchronized(Base.class) {
      // ...
      return format.parse(str);
    }
  }
}
```

3.3.2 Compliant Solution (Class Name Qualification)

In this compliant solution, the class name providing the lock (`Base`) is fully qualified.

```
class Base {
  static DateFormat format =
    DateFormat.getDateInstance(DateFormat.MEDIUM);

  public Date parse(String str) throws ParseException {
    synchronized (Base.class) {
      return format.parse(str);
    }
  }
}

// ...
```

This code example always synchronizes on the `Base.class` object, even if it is called from a `Derived` object.

3.3.3 Compliant Solution (`Class.forName()`)

This compliant solution uses the `Class.forName()` method to synchronize on the `Base` class's `Class` object.

```
class Base {
  static DateFormat format =
    DateFormat.getDateInstance(DateFormat.MEDIUM);

  public Date parse(String str) throws ParseException {
    synchronized (Class.forName("Base")) {
      return format.parse(str);
    }
  }
}

// ...
```

It is important that untrusted inputs are not accepted as arguments while loading classes using `Class.forName()`. See guideline "SEC05-J. Do not expose standard APIs that use the immediate caller's class loader instance to untrusted code[4]" for more information.

3.3.4 Noncompliant Code Example (`getClass()` Lock Object, Inner Class)

This noncompliant code example synchronizes on the class object returned by `getClass()` in the `parse()` method of the `Base` class. The `Base` class also has a nested `Helper` class whose `doSomethingAndParse()` method incorrectly synchronizes on the value returned by `getClass()`.

```
class Base {
  static DateFormat format =
    DateFormat.getDateInstance(DateFormat.MEDIUM);

  public Date parse(String str) throws ParseException {
    synchronized (getClass()) {
      return format.parse(str);
    }
  }

  public Date doSomething(String str) throws ParseException {
    return new Helper().doSomethingAndParse(str);
  }

  private class Helper {
    public Date doSomethingAndParse(String str) throws ParseException {
      synchronized(getClass()) { // Synchronizes on getClass()
        // ...
        return format.parse(str);
      }
    }
  }
}
```

The call to `getClass()` in the `Helper` class returns a `Helper` class object instead of the `Base` class object. Consequently, a thread that calls `Base.parse()` locks on a different object than a thread that calls `Base.doSomething()`. It is easy to overlook concurrency errors in inner classes because they exist within the body of the containing outer class. A reviewer might incorrectly assume that the two classes have the same locking strategy.

[4] This guideline is described at https://www.securecoding.cert.org/confluence/display/java/.

3.3.5 Compliant Solution (Class Name Qualification)

This compliant solution synchronizes using a `Base` class literal in the `parse()` and `doSomethingAndParse()` methods.

```
class Base {
  // ...

  public Date parse(String str) throws ParseException {
    synchronized (Base.class) {
      return format.parse(str);
    }
  }

  private class Helper {
    public Date doSomethingAndParse(String str) throws ParseException {
      synchronized(Base.class) { // Synchronizes on Base class literal
        // ...
        return format.parse(str);
      }
    }
  }
}
```

Consequently, both `Base` and `Helper` lock on `Base`'s intrinsic lock. Similarly, the `Class.forname()` method can be used instead of a class literal.

3.3.6 Risk Assessment

Synchronizing on the class object returned by `getClass()` can result in nondeterministic behavior.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
LCK02- J	medium	probable	medium	P8	L2

3.3.7 References

[Findbugs 2008]

[Miller 2009] Locking

[Pugh 2008] "Synchronization"

[Sun 2009b]

3.4 LCK03-J. Do not synchronize on the intrinsic locks of high-level concurrency objects

It is inappropriate to lock on an object of a class that implements one or both of the following interfaces of the `java.util.concurrent.locks` package: `Lock` and `Condition`. Using the intrinsic locks of these classes is a questionable practice even though the code may appear to function correctly. This problem is commonly discovered when code is refactored from intrinsic locking to the `java.util.concurrent` dynamic-locking utilities.

3.4.1 Noncompliant Code Example (`ReentrantLock` Lock Object)

The `doSomething()` method in this noncompliant code example synchronizes on the intrinsic lock of an instance of `ReentrantLock` instead of the reentrant mutual exclusion `Lock` encapsulated by `ReentrantLock`.

```
private final Lock lock = new ReentrantLock();

public void doSomething() {
  synchronized(lock) {
    // ...
  }
}
```

3.4.2 Compliant Solution (`lock()` and `unlock()`)

Instead of using the intrinsic locks of objects that implement the `Lock` interface, such as `ReentrantLock`, use the `lock()` and `unlock()` methods provided by the `Lock` interface.

```
private final Lock lock = new ReentrantLock();

public void doSomething() {
  lock.lock();
  try {
    // ...
  } finally {
    lock.unlock();
  }
}
```

If there is no requirement for using the advanced functionality of the `java.util.concurrent` package's dynamic-locking utilities, it is better to use the `Executor` framework or other concurrency primitives such as synchronization and atomic classes.

3.4.3 Risk Assessment

Synchronizing on the intrinsic lock of high-level concurrency utilities can cause nondeterministic behavior because the class can end up with two different locking policies.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
LCK03-J	medium	probable	medium	P8	L2

3.4.4 References

[Findbugs 2008]

[Miller 2009] Locking

[Pugh 2008] “Synchronization”

[Sun 2009b]

[Sun 2008a] Wrapper Implementations

3.5 LCK04-J. Do not synchronize on a collection view if the backing collection is accessible

The `java.util.Collections` interface's documentation [Sun 2009b] warns about the consequences of failing to synchronize on an accessible collection object when iterating over its view:

> *It is imperative that the user manually synchronize on the returned map when iterating over any of its collection views. . . Failure to follow this advice may result in non-deterministic behavior.*

A class that uses a collection view instead of the backing collection as the lock object may end up with two different locking strategies. In this case, if the backing collection is accessible to multiple threads, the class is not thread-safe.

3.5.1 Noncompliant Code Example (Collection View)

This noncompliant code example creates two views: a synchronized view of an empty `HashMap` encapsulated by the `map` field and a set view of the map's keys encapsulated by the `set` field. This example synchronizes on the `set` view [Sun 2008a].

```
// map has package-private accessibility
final Map<Integer, String> map =
    Collections.synchronizedMap(new HashMap<Integer, String>());
private final Set<Integer> set = map.keySet();

public void doSomething() {
  synchronized(set) {  // Incorrectly synchronizes on set
    for (Integer k : set) {
      // ...
    }
  }
}
```

In this example, `HashMap` provides the backing collection for `Map`, which provides the backing collection for `Set`, as shown in Figure 4:

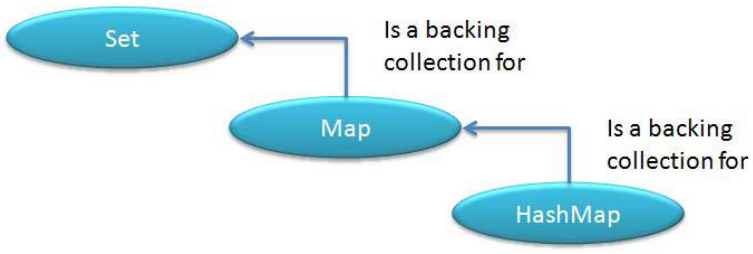

Figure 4: How Backing Collection Works in the Collection View, Noncompliant Code Example

`HashMap` is not accessible, but the `map` view is. Because the `set` view is synchronized instead of the `map` view, another thread can modify the contents of `map` and invalidate the `k` iterator.

3.5.2 Compliant Solution (Collection Lock Object)

This compliant solution synchronizes on the `map` view instead of the `set` view.

```
// map has package-private accessibility
final Map<Integer, String> map =
  Collections.synchronizedMap(new HashMap<Integer, String>());
private final Set<Integer> set = map.keySet();

public void doSomething() {
  synchronized(map) {  // Synchronize on map, not set
    for (Integer k : set) {
      // ...
    }
  }
}
```

This code is compliant because the map's underlying structure cannot be changed when an iteration is in progress.

3.5.3 Risk Assessment

Synchronizing on a collection view instead of the collection object can cause nondeterministic behavior.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
LCK04-J	low	probable	medium	P8	L2

3.5.4 References

[Sun 2009b] Class Collections

[Sun 2008a] Wrapper Implementations

3.6 LCK05-J. Synchronize access to static fields that may be modified by untrusted code

Methods that can be invoked from untrusted code to modify a static field must synchronize access to that field. That is necessary because there is no guarantee that untrusted clients will externally synchronize when accessing the field. Because a static field is shared by all clients, untrusted clients may violate the contract by failing to provide suitable locking.

According to Bloch [Bloch 2008]

> *If a method modifies a static field, you must synchronize access to this field, even if the method is typically used only by a single thread. It is not possible for clients to perform external synchronization on such a method because there can be no guarantee that unrelated clients will do likewise.*

Documented design intent is irrelevant when dealing with untrusted code because an attacker can always choose to ignore the documentation.

3.6.1 Noncompliant Code Example

This noncompliant code example does not synchronize access to the static `counter` field.

```
/** This class is not thread-safe */
public final class CountHits {
  private static int counter;

  public void incrementCounter() {
    counter++;
  }
}
```

This class definition does not violate guideline “VNA02-J. Ensure that compound operations on shared variables are atomic” on page 16, which only applies to classes that promise thread-safety. However, this class has a mutable static `counter` field that is modified by the publicly accessible `incrementCounter()` method. Consequently, this class cannot be used securely by trusted client code, if untrusted code can purposely fail to externally synchronize access to the field.

3.6.2 Compliant Solution

This compliant solution uses a static private final lock to protect the `counter` field and, consequently, does not depend on any external synchronization. This solution also complies with guideline "LCK00-J. Use private final lock objects to synchronize classes that may interact with untrusted code" on page 41.

```
/** This class is thread-safe */
public final class CountHits {
  private static int counter;
  private static final Object lock = new Object();

  public void incrementCounter() {
    synchronized (lock) {
      counter++;
    }
  }
}
```

3.6.3 Risk Assessment

Failing to internally synchronize access to static fields that may be modified by untrusted code will result in incorrectly synchronized code, if the author of the untrusted code chooses to ignore the synchronization policy.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
LCK05- J	low	probable	medium	P4	L3

3.6.4 References

[Bloch 2008] Item 67: "Avoid excessive synchronization"

[Sun 2009b]

3.7 LCK06-J. Do not use an instance lock to protect shared static data

Shared static data should not be protected using instance locks because they are ineffective when two or more instances of the class are created. Consequently, shared state is not safe for concurrent access unless a static lock object is used. If the class can interact with untrusted code, the lock must also be private and final, as per guideline "LCK00-J. Use private final lock objects to synchronize classes that may interact with untrusted code" on page 41.

3.7.1 Noncompliant Code Example (Non-Static Lock Object for Static Data)

This noncompliant code example uses a non-static lock object to guard access to a static `counter` field. If two `Runnable` tasks are started, they will create two instances of the lock object and lock on each one separately.

```
public final class CountBoxes implements Runnable {
  private static volatile int counter;
  // ...
  private final Object lock = new Object();

  @Override public void run() {
    synchronized(lock) {
      counter++;
      // ...
    }
  }

  public static void main(String[] args) {
    for(int i = 0; i < 2; i++) {
      new Thread(new CountBoxes()).start();
    }
  }
}
```

This example does not prevent either thread from observing an inconsistent value of `counter` because the increment operation on volatile fields is non-atomic in the absence of proper synchronization (see guideline "VNA02-J. Ensure that compound operations on shared variables are atomic" on page 16).

3.7.2 Noncompliant Code Example (Method Synchronization for Static Data)

This noncompliant code example uses method synchronization to protect access to the static class `counter` field.

```
public final class CountBoxes implements Runnable {
  private static volatile int counter;
  // ...

  public synchronized void run() {
    counter++;
    // ...
  }
  // ...
}
```

In this case, the intrinsic lock is associated with each instance of the class and not with the class itself. Consequently, threads constructed using different `Runnable` instances may observe inconsistent values of `counter`.

3.7.3 Compliant Solution (Static Lock Object)

This compliant solution declares the lock object static and consequently ensures the atomicity of the increment operation.

```
public class CountBoxes implements Runnable {
  private static int counter;
  // ...
  private static final Object lock = new Object();

  public void run() {
    synchronized(lock) {
      counter++;
      // ...
  }
  // ...
}
```

There is no need to declare the `counter` variable volatile when using synchronization.

3.7.4 Risk Assessment

Using an instance lock to protect shared static data can result in nondeterministic behavior.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
LCK06- J	medium	probable	medium	P8	L2

3.7.5 References

[Sun 2009b]

3.8 LCK07-J. Avoid deadlock by requesting and releasing locks in the same order

To avoid data corruption in multithreaded Java programs, shared data must be protected from concurrent modifications and accesses. Locking can be performed at the object level using synchronized methods, synchronized blocks, or the `java.util.concurrent` dynamic, lock objects. However, excessive use of locking can result in deadlock.

Java does not prevent deadlock or require its detection [Gosling 2005]. Deadlock can occur when two or more threads request and release locks in different orders. Consequently, it can be avoided by acquiring and releasing locks in the same order.

Additionally, synchronization should be limited to cases where it is absolutely necessary. For example, the `paint()`, `dispose()`, `stop()`, and `destroy()` methods should never be synchronized in an applet because they are always called and used from dedicated threads. The `Thread.stop()` and `Thread.destroy()` methods are deprecated. For more information, see guideline “THI05-J. Do not use Thread.stop() to terminate threads” on page 110.

This guideline also applies to programs that need to work with a limited set of resources. For example, liveness issues can arise when two or more threads are waiting for each other to release resources such as database connections. These issues can be resolved by letting each waiting thread retry the operation at random intervals, until they succeed in acquiring the resource successfully.

3.8.1 Noncompliant Code Example (Different Lock Orders)

This noncompliant code example can deadlock because of excessive synchronization. The `balanceAmount` field represents the total balance amount available for a particular `BankAccount` object. A user is allowed to initiate an operation that atomically transfers a specified amount from one account to another.

```
final class BankAccount {
  private double balanceAmount;  // Total amount in bank account

  BankAccount(double balance) {
    this.balanceAmount = balance;
  }

  // Deposits the amount from this object instance to BankAccount instance argument ba
  private void depositAmount(BankAccount ba, double amount) {
    synchronized (this) {
      synchronized(ba) {
        if(amount > balanceAmount) {
          throw new IllegalArgumentException("Transfer cannot be completed");
        }
        ba.balanceAmount += amount;
        this.balanceAmount -= amount;
      }
    }
  }
```

```
  public static void initiateTransfer(final BankAccount first,
    final BankAccount second, final double amount) {

    Thread transfer = new Thread(new Runnable() {
      public void run() {
        first.depositAmount(second, amount);
      }
    });
    transfer.start();
  }
}
```

Objects of this class are prone to deadlock. An attacker that has two bank accounts can construct two threads that initiate balance transfers from two different `BankAccount` object instances, `a` and `b`. For example, consider the following code:

```
BankAccount a = new BankAccount(5000);
BankAccount b = new BankAccount(6000);
initiateTransfer(a, b, 1000); // starts thread 1
initiateTransfer(b, a, 1000); // starts thread 2
```

Each transfer is performed in its own thread. The first thread atomically transfers the amount from `a` to `b` by depositing it in account `b` and then withdrawing the same amount from `a`. The second thread performs the reverse operation; that is, it transfers the amount from `b` to `a`. When executing `depositAmount()`, the first thread acquires a lock on object `a`. The second thread could acquire a lock on object `b` before the first thread can. Subsequently, the first thread would request a lock on `b`, which is already held by the second thread. The second thread would request a lock on `a`, which is already held by the first thread. This constitutes a deadlock condition, because neither thread can proceed.

This noncompliant code example may or may not cause deadlock, depending on the scheduling details of the platform. Deadlock will occur when two threads request the same two locks in different orders and each thread obtains a lock that prevents the other thread from completing its transfer. Deadlock will not occur when two threads request the same two locks but one thread completes its transfer before the other thread begins. Similarly, deadlock will not occur if the two threads request the same two locks in the same order (which would happen if they both transfer money from one account to a second account) or if two transfers involving distinct accounts occur concurrently.

3.8.2 Compliant Solution (Private Static Final Lock Object)

Deadlock can be avoided by synchronizing on a private static final lock object before performing any account transfers.

```
final class BankAccount {
  private double balanceAmount;  // Total amount in bank account
  private static final Object lock = new Object();

  BankAccount(double balance) {
```

```
    this.balanceAmount = balance;
  }

  // Deposits the amount from this object instance to BankAccount instance argument ba
  private void depositAmount(BankAccount ba, double amount) {
    synchronized (lock) {
      if (amount > balanceAmount) {
        throw new IllegalArgumentException("Transfer cannot be completed");
      }
      ba.balanceAmount += amount;
      this.balanceAmount -= amount;
    }
  }

  public static void initiateTransfer(final BankAccount first,
    final BankAccount second, final double amount) {

    Thread transfer = new Thread(new Runnable() {
      @Override public void run() {
        first.depositAmount(second, amount);
      }
    });
    transfer.start();
  }
}
```

In this scenario, deadlock cannot occur when two threads with two different `BankAccount` objects try to transfer to each others' accounts simultaneously. One thread will acquire the private lock, complete its transfer, and release the lock before the other thread can proceed.

This solution comes with a performance penalty because a `private static` lock restricts the system to performing only one transfer at a time. Two transfers involving four distinct accounts (with distinct target accounts) cannot be performed concurrently. This penalty increases considerably as the number of `BankAccount` objects increase. Consequently, this solution does not scale well.

3.8.3 Compliant Solution (Ordered Locks)

This compliant solution ensures that multiple locks are acquired and released in the same order. It requires that an ordering over `BankAccount` objects is available. The ordering is enforced by having the `BankAccount` class implement the `java.lang.Comparable` interface and override the `compareTo()` method.

```
final class BankAccount implements Comparable<BankAccount> {
  private double balanceAmount;  // Total amount in bank account
  private final Object lock;

  private final long id; // Unique for each BankAccount
  private static long NextID = 0; // Next unused ID
```

```
  BankAccount(double balance) {
    this.balanceAmount = balance;
    this.lock = new Object();
    this.id = this.NextID++;
  }

  @Override public int compareTo(BankAccount ba) {
     return (this.id > ba.id) ? 1 : (this.id < ba.id) ? -1 : 0;
  }

  // Deposits the amount from this object instance to BankAccount instance argument ba
  public void depositAmount(BankAccount ba, double amount) {
    BankAccount former, latter;
    if (compareTo(ba) < 0) {
      former = this;
      latter = ba;
    } else {
      former = ba;
      latter = this;
    }
    synchronized (former) {
      synchronized (latter) {
        if (amount > balanceAmount) {
          throw new IllegalArgumentException("Transfer cannot be completed");
        }
        ba.balanceAmount += amount;
        this.balanceAmount -= amount;
      }
    }
  }

  public static void initiateTransfer(final BankAccount first,
    final BankAccount second, final double amount) {

    Thread transfer = new Thread(new Runnable() {
      @Override public void run() {
        first.depositAmount(second, amount);
      }
    });
    transfer.start();
  }
}
```

Whenever a transfer occurs, the two `BankAccount` objects are ordered so that the `first` object's lock is acquired before the `second` object's lock. Consequently, if two threads attempt transfers between the same two accounts, they will both try to acquire the first account's lock before the second's. As a result, one thread will acquire both locks, complete the transfer, and release both locks before the other thread can proceed.

Unlike in the previous compliant solution, multiple transfers can happen concurrently, as long as they involve distinct target accounts.

3.8.4 Compliant Solution (`ReentrantLock`)

In this compliant solution, each `BankAccount` has an associated `java.util.concurrent.locks.ReentrantLock`. This design permits the `depositAmount()` method to try to acquire the locks of both accounts and to release the locks if it fails and try again later.

```
final class BankAccount {
  private double balanceAmount;  // Total amount in bank account
  private final Lock lock = new ReentrantLock();
  private final Random number = new Random(123L);

  BankAccount(double balance) {
    this.balanceAmount = balance;
  }

  // Deposits amount from this object instance to BankAccount instance argument ba
  private void depositAmount(BankAccount ba, double amount)
                                                  throws InterruptedException {
    while (true) {
      if (this.lock.tryLock()) {
        try {
          if (ba.lock.tryLock()) {
            try {
              if (amount > balanceAmount) {
                throw new IllegalArgumentException("Transfer cannot be completed");
              }
              ba.balanceAmount += amount;
              this.balanceAmount -= amount;
              break;
            } finally {
              ba.lock.unlock();
            }
          }
        } finally {
          this.lock.unlock();
        }
      }
      int n = number.nextInt(1000);
      int TIME = 1000 + n; // 1 second + random delay to prevent livelock
      Thread.sleep(TIME);
    }
  }

  public static void initiateTransfer(final BankAccount first,
    final BankAccount second, final double amount) {
```

```
    Thread transfer = new Thread(new Runnable() {
      public void run() {
        try {
          first.depositAmount(second, amount);
        } catch (InterruptedException e) {
          Thread.currentThread().interrupt(); // Reset interrupted status
        }
      }
    });
    transfer.start();
  }
}
```

Deadlock is impossible in this compliant solution because no method grabs a lock and holds it indefinitely. If the current object's lock is acquired but the second lock is unavailable, the first lock is released and the thread sleeps for some specified amount of time before attempting to reacquire the lock.

Code that uses this lock has behavior similar to that of synchronized code that uses the traditional monitor lock. `ReentrantLock` provides several other capabilities. For example, the `tryLock()` method does not block waiting, if another thread is already holding the lock. The `java.util.concurrent.locks.ReentrantReadWriteLock` class can be used when some threads require a lock to write information, while other threads require the lock to concurrently read the information.

3.8.5 Noncompliant Code Example (Different Lock Orders, Recursive)

The following immutable `WebRequest` class encapsulates a web request received by a server:

```
// Immutable WebRequest
public final class WebRequest {
  private final long bandwidth;
  private final long responseTime;

  public WebRequest(long bandwidth, long responseTime) {
    this.bandwidth = bandwidth;
    this.responseTime = responseTime;
  }

  public long getBandwidth() {
    return bandwidth;
  }

  public long getResponseTime() {
    return responseTime;
  }
}
```

Each request has a response time associated with it, along with a measurement of the network bandwidth required to fulfill the request.

This noncompliant code example monitors web requests and provides routines for calculating the average bandwidth and response time required to service incoming requests.

```
public final class WebRequestAnalyzer {
  private final Vector<WebRequest> requests = new Vector<WebRequest>();

  public boolean addWebRequest(WebRequest request) {
    return requests.add(new WebRequest(request.getBandwidth(),
                        request.getResponseTime()));
  }

  public double getAverageBandwidth() {
    if (requests.size() == 0) {
      throw new IllegalStateException("The vector is empty!");
    }
    return calculateAverageBandwidth(0, 0);
  }

  public double getAverageResponseTime() {
    if (requests.size() == 0) {
      throw new IllegalStateException("The vector is empty!");
    }
    return calculateAverageResponseTime(requests.size() - 1, 0);
  }

  private double calculateAverageBandwidth(int i, long bandwidth) {
    if (i == requests.size()) {
      return bandwidth / requests.size();
    }
    synchronized (requests.elementAt(i)) {
      bandwidth += requests.get(i).getBandwidth();
      // Acquires locks in increasing order
      return calculateAverageBandwidth(++i, bandwidth);
    }
  }

  private double calculateAverageResponseTime(int i, long responseTime) {
    if (i <= -1) {
      return responseTime / requests.size();
    }
```

```
    synchronized (requests.elementAt(i)) {
      responseTime += requests.get(i).getResponseTime();
      // Acquires locks in decreasing order
      return calculateAverageResponseTime(--i, responseTime);
    }
  }
}
```

The monitoring application is built around the `WebRequestAnalyzer` class that maintains a list of web requests using the `requests` vector and includes the `addWebRequest()` setter method. Any thread can request the average bandwidth or average response time of all web requests by invoking the `getAverageBandwidth()` and `getAverageResponseTime()` methods.

These methods use fine-grained locking by holding locks on individual elements (web requests) of the vector. These locks permit new requests to be added while the computations are still underway. Consequently, the statistics reported by the methods are accurate when they return the results.

Unfortunately, this noncompliant code example is prone to deadlock because the recursive calls within the synchronized regions of these methods acquire the intrinsic locks in opposite numerical orders. That is, `calculateAverageBandwidth()` requests locks from index 0 up to `requests.size()` - 1, whereas `calculateAverageResponseTime()` requests them from index `requests.size()` - 1 down to 0. Because of recursion, no previously acquired locks are released by either method. Deadlock occurs when two threads call these methods out of order, because one thread calls `calculateAverageBandwidth()`, while the other calls `calculateAverageResponseTime()` before either method has finished executing.

For example, if there are 20 requests in the vector and one thread calls `getAverageBandwidth()`, the thread acquires the intrinsic lock of `WebRequest` 0, the first element in the vector. Meanwhile, if a second thread calls `getAverageResponseTime()`, it acquires the intrinsic lock of web request 19, the last element in the vector. Consequently, deadlock results because neither thread can acquire all of the locks and proceed with the calculations.

Note that the `addWebRequest()` method also has a race condition with `calculateAverageResponseTime()`. While iterating over the vector, new elements can be added to the vector, invalidating the results of the previous computation. This race condition can be prevented by locking on the last element of the vector (when it contains at least one element) before inserting the element.

3.8.6 Compliant Solution

In this compliant solution, the two calculation methods acquire and release locks in the same order, beginning with the first web request in the vector.

```
public final class WebRequestAnalyzer {
  private final Vector<WebRequest> requests = new Vector<WebRequest>();

  public boolean addWebRequest(WebRequest request) {
    return requests.add(new WebRequest(request.getBandwidth(),
                        request.getResponseTime()));
  }

  public double getAverageBandwidth() {
    if (requests.size() == 0) {
      throw new IllegalStateException("The vector is empty!");
    }
    return calculateAverageBandwidth(0, 0);
  }

  public double getAverageResponseTime() {
    if (requests.size() == 0) {
      throw new IllegalStateException("The vector is empty!");
    }
    return calculateAverageResponseTime(0, 0);
  }

  private double calculateAverageBandwidth(int i, long bandwidth) {
    if (i == requests.size()) {
      return bandwidth / requests.size();
    }
    synchronized (requests.elementAt(i)) { // Acquires locks in increasing order
      bandwidth += requests.get(i).getBandwidth();
      return calculateAverageBandwidth(++i, bandwidth);
    }
  }

  private double calculateAverageResponseTime(int i, long responseTime) {
    if (i == requests.size()) {
      return responseTime / requests.size();
    }
    synchronized (requests.elementAt(i)) {
      responseTime += requests.get(i).getResponseTime();
      // Acquires locks in increasing order
      return calculateAverageResponseTime(++i, responseTime);
    }
  }
}
```

Consequently, while one thread is calculating the average bandwidth or response time, another thread cannot interfere or induce deadlock. That is because the other thread first needs to synchronize on the first web request, which cannot happen before the first calculation completes.

There is no need to lock on the last element of the vector in `addWebRequest()` for two reasons: (1) because locks are acquired in increasing order in all the methods and (2) because updates to the vector are reflected in the results of the computations.

3.8.7 Risk Assessment

Acquiring and releasing locks in the wrong order can result in deadlock.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
LCK07- J	low	likely	high	P3	L3

3.8.8 References

[Gosling 2005] Chapter 17, “Threads and Locks”

[Halloway 2000]

[MITRE 2010] CWE ID 412, “Unrestricted Lock on Critical Resource”

3.9 LCK08-J. Ensure actively held locks are released on exceptional conditions

An exceptional condition can circumvent the release of a lock, leading to deadlock. According to the Java API [Sun 2009b]

> *A `ReentrantLock` is owned by the thread last successfully locking, but not yet unlocking it. A thread invoking `lock` will return, successfully acquiring the lock, when the lock is not owned by another thread.*

Consequently, an unreleased lock in any thread will stop other threads from acquiring the same lock. Intrinsic locks of class objects used for method and block synchronization are automatically released on exceptional conditions (such as abnormal thread termination).

3.9.1 Noncompliant Code Example (Checked Exception)

This noncompliant code example protects a resource using a `ReentrantLock` but fails to release the lock if an exception occurs while performing operations on the open file. If an exception is thrown, control transfers to the `catch` block, and the call to `unlock()` is not executed.

```
public final class Client {
  public void doSomething(File file) {
    final Lock lock = new ReentrantLock();
    try {
      lock.lock();
      InputStream in = new FileInputStream(file);
      // Perform operations on the open file
      lock.unlock();
    } catch (FileNotFoundException fnf) {
      // Handle the exception
    }
  }
}
```

Note that the lock is not released, even when the `doSomething()` method returns.

This noncompliant code example does not close the input stream and, consequently, also violates guideline "FIO06-J. Ensure all resources are properly closed when they are no longer needed."[5]

3.9.2 Compliant Solution (`finally` Block)

This compliant solution encapsulates operations that may throw an exception in a `try` block immediately after acquiring the lock. The lock is acquired just before the `try` block, which guarantees that the lock is held when the `finally` block executes. Invoking `Lock.unlock()` in the `finally` block ensures that the lock is released, regardless of whether an exception occurred.

```
public final class Client {
  public void doSomething(File file) {
    final Lock lock = new ReentrantLock();
```

[5] This guideline is described at https://www.securecoding.cert.org/confluence/display/java/.

```
    InputStream in = null;
    lock.lock();
    try {
      in = new FileInputStream(file);
      // Perform operations on the open file
    } catch(FileNotFoundException fnf) {
      // Forward to handler
    } finally {
      lock.unlock();

      if(in != null) {
        try {
           in.close();
         } catch (IOException e) {
           // Forward to handler
        }
      }
    }
  }
}
```

3.9.3 Compliant Solution (Execute-Around Idiom)

The execute-around idiom provides a generic mechanism for performing resource allocation and clean-up operations so that the client can focus on specifying only the required functionality. This idiom reduces clutter in client code and provides a secure mechanism for resource management.

In this compliant solution, the client's `doSomething()` method provides only the required functionality by implementing the `doSomethingWithFile()` method of the `LockAction` interface, without having to manage the acquisition and release of locks or the open and close operations of files. The `ReentrantLockAction` class encapsulates all resource management actions.

```
public interface LockAction {
  void doSomethingWithFile(InputStream in);
}

public final class ReentrantLockAction {
  public static void doSomething(File file, LockAction action)  {
    Lock lock = new ReentrantLock();
    InputStream in = null;
    lock.lock();
    try {
      in = new FileInputStream(file);
      action.doSomethingWithFile(in);
    } catch (FileNotFoundException fnf) {
      // Forward to handler
    } finally {
      lock.unlock();
```

```
      if (in != null) {
        try {
          in.close();
        } catch (IOException e) {
          // Forward to handler
        }
      }
    }
  }
}

public final class Client {
  public void doSomething(File file) {
    ReentrantLockAction.doSomething(file, new LockAction() {
      public void doSomethingWithFile(InputStream in) {
        // Perform operations on the open file
      }
    });
  }
}
```

3.9.4 Noncompliant Code Example (Unchecked Exception)

This noncompliant code example uses a `ReentrantLock` to protect a `java.util.Date` instance, which is not thread-safe by design. The `doSomethingSafely()` method must catch `Throwable` to comply with guideline "EXC06-J. Do not allow exceptions to transmit sensitive information."[6]

```
final class DateHandler {
  private final Date date = new Date();
  final Lock lock = new ReentrantLock();
  public void doSomethingSafely(String str) {
    try {
      doSomething(str);
    } catch(Throwable t) {
      // Forward to handler
    }
  }
  public void doSomething(String str) {
    lock.lock();
    String dateString = date.toString();
    if (str.equals(dateString)) {
      // ...
    }
    lock.unlock();
  }
}
```

[6] This guideline is described at https://www.securecoding.cert.org/confluence/display/java/.

Because the `doSomething()` method fails to check if `str` is null, a runtime exception can occur, preventing the lock from being released.

3.9.5 Compliant Solution (`finally` Block)

This compliant solution encapsulates all operations that can throw an exception in a `try` block and releases the lock in the associated `finally` block.

```
final class DateHandler {
  private final Date date = new Date();
  final Lock lock = new ReentrantLock();

  public void doSomethingSafely(String str) {
    try {
      doSomething(str);
    } catch(Throwable t) {
      // Forward to handler
    }
  }

  public void doSomething(String str) {
    lock.lock();
    try {
      String dateString = date.toString();
      if (str != null && str.equals(dateString)) {
        // ...
      }
    } finally {
      lock.unlock();
    }
  }
}
```

Consequently, the lock is released even in the event of a runtime exception. The `doSomething()` method also ensures that the string is not null to avoid throwing a `NullPointerException`.

3.9.6 Risk Assessment

Failing to release locks on exceptional conditions may lead to thread starvation and deadlock.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
LCK08- J	low	likely	low	P9	L2

3.9.7 References

[Sun 2009b] Class `ReentrantLock`

3.10 LCK09-J. Do not perform operations that may block while holding a lock

Holding locks while performing time-consuming or blocking operations can severely degrade system performance and result in starvation. Furthermore, deadlock can result if interdependent threads block indefinitely. Blocking operations include network, file, and console I/O (for example, `Console.readLine()`) and object serialization. Deferring a thread indefinitely also constitutes a blocking operation.

If the JVM interacts with a file system that operates over an unreliable network, file I/O might incur a large performance penalty. In such cases, avoid file I/O over the network when holding a lock. File operations (such as logging) that may block waiting for the output stream lock or for I/O to complete may be performed in a dedicated thread to speed up task processing. Logging requests can be added to a queue, given that the queue's `put()` operation incurs little overhead as compared to file I/O [Goetz 2006].

3.10.1 Noncompliant Code Example (Deferring a Thread)

This noncompliant code example defines a utility method that accepts a `time` argument.

```
public synchronized void doSomething(long time)
    throws InterruptedException {
  // ...
  Thread.sleep(time);
}
```

Because the method is synchronized, if the thread is suspended, other threads are unable to use the synchronized methods of the class. The current object's monitor is not released because the `Thread.sleep()` method does not have any synchronization semantics, as detailed in guideline "THI00-J. Do not assume that the `sleep()`, `yield()`, or `getState()` methods provide synchronization semantics" on page 91.

3.10.2 Compliant Solution (Intrinsic Lock)

This compliant solution defines the `doSomething()` method with a `timeout` parameter instead of the `time` value. Using the `Object.wait()` rather than the `Thread.sleep()` method allows setting a timeout period during which a notification may awaken the thread.

```
public synchronized void doSomething(long timeout)
    throws InterruptedException {

  while (<condition does not hold>) {
    wait(timeout); // Immediately leaves current monitor
  }
}
```

The current object's monitor is released immediately upon entering the wait state. After the timeout period has elapsed, the thread resumes execution after reacquiring the current object's monitor.

According to the Java API class `Object` documentation [Sun 2009b]

> *Note that the `wait` method, as it places the current thread into the wait set for this object, unlocks only this object; any other objects on which the current thread may be synchronized remain locked while the thread waits. This method should only be called by a thread that is the owner of this object's monitor.*

Ensure that a thread that holds locks on other objects releases them appropriately, before entering the wait state. Additional guidance on waiting and notification is available in guidelines “THI03-J. Always invoke wait() and await() methods inside a loop” on page 101 and “THI04-J. Notify all waiting threads instead of a single thread” on page 104.

3.10.3 Noncompliant Code Example (Network I/O)

This noncompliant code example shows the `sendPage()`method that sends a `Page` object from a server to a client. The method is synchronized so that the `pageBuff` array is accessed safely when multiple threads request concurrent access.

```
// Class Page is defined separately. It stores and returns the Page name via getName()
Page[] pageBuff = new Page[MAX_PAGE_SIZE];

public synchronized boolean sendPage(Socket socket, String pageName)
    throws IOException {
  // Get the output stream to write the Page to
  ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());

  // Find the Page requested by the client (this operation requires synchronization)
  Page targetPage = null;
  for (Page p : pageBuff) {
    if (p.getName().compareTo(pageName) == 0) {
      targetPage = p;
    }
  }

  // Requested Page does not exist
  if (targetPage == null) {
    return false;
  }

  // Send the Page to the client (does not require any synchronization)
  out.writeObject(targetPage);

  out.flush();
  out.close();
  return true;
}
```

Calling `writeObject()` within the synchronized `sendPage()` method can result in delays and deadlock-like conditions in high-latency networks or when network connections are inherently lossy.

3.10.4 Compliant Solution

This compliant solution separates the process into a sequence of steps:

1. Perform actions on data structures requiring synchronization.
2. Create copies of the objects to be sent.
3. Perform network calls in a separate method that does not require any synchronization.

In this compliant solution, the synchronized `getPage()`method is called from the unsynchronized `sendPage()` method to retrieve the requested `Page` in the `pageBuff` array. After the `Page` is retrieved, `sendPage()` calls the unsynchronized `deliverPage()`method to deliver the `Page` to the client.

```
public boolean sendPage(Socket socket, String pageName) { // No synchronization
  Page targetPage = getPage(pageName);

  if (targetPage == null)
    return false;

  return deliverPage(socket, targetPage);
}

private synchronized Page getPage(String pageName) { // Requires synchronization
  Page targetPage = null;

  for (Page p : pageBuff) {
    if (p.getName().equals(pageName)) {
      targetPage = p;
    }
  }
  return targetPage;
}

// Return false if an error occurs, true if successful
public boolean deliverPage(Socket socket, Page page) {
  ObjectOutputStream out = null;
  boolean result = true;
  try {
    // Get the output stream to write the Page to
    out = new ObjectOutputStream(socket.getOutputStream());

    // Send the Page to the client
    out.writeObject(page);
  } catch (IOException io) {
    result = false;
  } finally {
    if (out != null) {
      try {
        out.flush();
```

```
        out.close();
      } catch (IOException e) {
        result = false;
      }
    }
  }
  return result;
}
```

3.10.5 Exceptions

LCK09-EX1: Classes that provide an appropriate termination mechanism to callers are allowed to violate this guideline (see guideline "THI06-J. Ensure that threads and tasks performing blocking operations can be terminated" on page 114).

LCK09-EX2: A method that requires multiple locks may hold several locks while waiting for the remaining locks to become available. This constitutes a valid exception, although the programmer must follow other applicable guidelines to avoid deadlock. See guideline "LCK07-J. Avoid deadlock by requesting and releasing locks in the same order" on page 63 for more information.

3.10.6 Risk Assessment

Blocking or lengthy operations performed within synchronized regions may result in a deadlocked or unresponsive system.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
LCK09-J	low	probable	high	P2	L3

3.10.7 References

[Gosling 2005] Chapter 17, "Threads and Locks"

[Grosso 2001] Chapter 10, "Serialization"

[Rotem-Gal-Oz 2008] "Falacies of Distributed Computing Explained"

[Sun 2009b] Class `Object`

3.11 LCK10-J. Do not use incorrect forms of the double-checked locking idiom

Instead of initializing a member object using a constructor, *lazy initialization* can be used to defer the construction of the member object until an instance is actually required. Lazy initialization also helps in breaking harmful circularities in class and instance initialization, and performing other optimizations [Bloch 2005a].

A class or an instance method is used for lazy initialization, depending on whether the member object is static. The method checks whether the instance has already been created and, if not, creates it. If the instance already exists, it simply returns it:

```
// Correct single threaded version using lazy initialization
final class Foo {
  private Helper helper = null;

  public Helper getHelper() {
    if (helper == null) {
      helper = new Helper();
    }
    return helper;
  }
  // ...
}
```

In a multithreaded application, initialization must be synchronized so that multiple threads do not create extraneous instances of the member object:

```
// Correct multithreaded version using synchronization
final class Foo {
  private Helper helper = null;

  public synchronized Helper getHelper() {
    if (helper == null) {
      helper = new Helper();
    }
    return helper;
  }
  // ...
}
```

The double-checked locking idiom improves performance by limiting synchronization to the rare case of new instance creation and foregoing it during the common case of retrieving an already created instance.

Incorrect forms of the double-checked idiom include those that allow an uninitialized or partially initialized object to be published.

3.11.1 Noncompliant Code Example

The double-checked locking pattern uses block synchronization instead of method synchronization and installs an additional `null` check before attempting synchronization. This noncompliant code example uses the incorrect form of the double-checked locking idiom.

```
// "Double-Checked Locking" idiom
final class Foo {
  private Helper helper = null;
  public Helper getHelper() {
    if (helper == null) {
      synchronized (this) {
        if (helper == null) {
          helper = new Helper();
        }
      }
    }
    return helper;
  }
  // Other methods and members...
}
```

According to Pugh [Pugh 2004]

> *. . . writes that initialize the `Helper` object and the write to the `helper` field can be done or perceived out of order. As a result, a thread which invokes `getHelper()` could see a non-null reference to a `helper` object, but see the default values for fields of the `helper` object, rather than the values set in the constructor.*
>
> *Even if the compiler does not reorder those writes, on a multiprocessor the processor or the memory system may reorder those writes, as perceived by a thread running on another processor.*

Also see guideline “TSM03-J. Do not publish partially initialized objects” on page 162.

3.11.2 Compliant Solution (Volatile)

This compliant solution declares the `helper` field volatile.

```
// Works with acquire/release semantics for volatile
// Broken under JDK 1.4 and earlier
final class Foo {
  private volatile Helper helper = null;

  public Helper getHelper() {
    if (helper == null) {
      synchronized (this) {
        if (helper == null) {
          helper = new Helper(); // If the helper is null, create a new instance
        }
      }
```

```
    }
    return helper; // If helper is non-null, return its instance
  }
}
```

If a thread initializes the `Helper` object, a happens-before relationship is established between this thread and another that retrieves and returns the instance [Pugh 2004, Manson 2004].

3.11.3 Compliant Solution (Static Initialization)

This compliant solution initializes the `helper` field in the declaration of the static variable.[7]

```
final class Foo {
  private static final Helper helper = new Helper();

  public static Helper getHelper() {
    return helper;
  }
}
```

Variables that are declared static and initialized at declaration, or from a static initializer, are guaranteed to be fully constructed before being made visible to other threads.

3.11.4 Compliant Solution (Initialize-On-Demand, Holder Class Idiom)

This compliant solution uses the initialize-on-demand, holder class idiom that implicitly incorporates lazy initialization by declaring a static variable within the static `Holder` inner class.

```
final class Foo {
  // Lazy initialization
  private static class Holder {
    static Helper helper = new Helper();
  }

  public static Helper getInstance() {
    return Holder.helper;
  }
}
```

Initialization of the static `helper` field is deferred until the `getInstance()` method is called. This idiom is a better choice than the double-checked, locking idiom for lazily initializing static fields [Bloch 2008]. However, this idiom cannot be used to lazily initialize instance fields [Bloch 2001].

7 *The Java Memory Model: the building block of concurrency*, by Jeremy Manson. Java Developer Connection Tech Tips, by Glen McCluskey, April 10, 2001.

3.11.5 Compliant Solution (`ThreadLocal` Storage)

This compliant solution (originally suggested by Terekhov [Pugh 2004]) uses a `ThreadLocal` object to lazily create a `Helper` instance.

```
final class Foo {
  private final ThreadLocal<Foo> perThreadInstance = new ThreadLocal<Foo>();
  private Helper helper = null;

  public Helper getHelper() {
    if (perThreadInstance.get() == null) {
      createHelper();
    }
    return helper;
  }

  private synchronized void createHelper() {
    if (helper == null) {
      helper = new Helper();
    }
    // Any non-null value can be used as an argument to set()
    perThreadInstance.set(this);
  }
}
```

3.11.6 Compliant Solution (Immutable)

In this compliant solution, the `Helper` class is immutable and consequently guaranteed to be fully constructed before becoming visible. In this case, there are no further requirements to ensure that the double-checked locking idiom does not result in the publication of an uninitialized or partially initialized field.

```
public final class Helper {
  private final int n;

  public Helper(int n) {
    this.n = n;
  }

  // Other fields and methods, all fields are final
}

final class Foo {
  private Helper helper = null;

  public Helper getHelper() {
    if (helper == null) {
      synchronized (this) {
        if (helper == null) {
```

```
          helper = new Helper(42); // If the helper is null, create a new instance
        }
      }
    }
    return helper; // If helper is non-null, return its instance
  }
}
```

3.11.7 Exceptions

LCK10-EX1: The noncompliant form of the double-checked locking idiom can be used for 32-bit primitive values (for example, `int` or `float`) [Pugh 2004]. Note that it does not work for values of type `long` or `double` because unsynchronized reads/writes of 64-bit primitives are not guaranteed to be atomic (see guideline "VNA05-J. Ensure atomicity when reading and writing 64-bit values" on page 33).

3.11.8 Risk Assessment

Using incorrect forms of the double-checked, locking idiom can lead to synchronization problems.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
LCK10- J	low	probable	medium	P4	L3

3.11.9 References

[Bloch 2001] Item 48: "Synchronize access to shared mutable data"

[Bloch 2008] Item 71: "Use lazy initialization judiciously"

[Gosling 2005] Section 12.4, "Initialization of Classes and Interfaces"

[MITRE 2010] CWE ID 609 "Double-Checked Locking"

[Pugh 2004]

[Sun 2009b]

3.12 LCK11-J. Avoid client-side locking when using classes that do not commit to their locking strategy

According to Goetz and colleagues [Goetz 2006]

> *Client-side locking entails guarding client code that uses some object X with the lock X uses to guard its own state. In order to use client-side locking, you must know what lock X uses.*

While client-side locking is acceptable if the thread-safe class commits to its locking strategy and clearly documents it, Goetz and colleagues caution against its misuse [Goetz 2006]:

> *If extending a class to add another atomic operation is fragile because it distributes the locking code for a class over multiple classes in an object hierarchy, client-side locking is even more fragile because it entails putting locking code for class C into classes that are totally unrelated to C. Exercise care when using client-side locking on classes that do not commit to their locking strategy.*

The documentation of a class that supports client-side locking should explicitly state its applicability. For example, the `java.util.concurrent.ConcurrentHashMap<K,V>` class should not be used for client-side locking because its documentation states [Sun 2009b]

> *. . . even though all operations are thread-safe, retrieval operations do not entail locking, and there is not any support for locking the entire table in a way that prevents all access. This class is fully interoperable with `Hashtable` in programs that rely on its thread safety but not on its synchronization details.*

In general, use client-side locking only when the documentation of the class recommends it. For example, the documentation of the `synchronizedList()`wrapper method of the `java.util.Collections` class states [Sun 2009b]

> *In order to guarantee serial access, it is critical that all access to the backing list is accomplished through the returned list. It is imperative that the user manually synchronize on the returned list when iterating over it. Failure to follow this advice may result in nondeterministic behavior.*

When the backing list is inaccessible to an untrusted client, note that this advice is consistent with guideline “LCK04-J. Do not synchronize on a collection view if the backing collection is accessible” on page 57.

3.12.1 Noncompliant Code Example (Intrinsic Lock)

This noncompliant code example uses the thread-safe `Book` class that cannot be refactored. Refactoring might be impossible, for example, if the source code is not available for review or the class is part of a general library that cannot be extended.

```
final class Book {
  // May change its locking policy in the future to use private final locks
  private final String title;
  private Calendar dateIssued;
  private Calendar dateDue;

  Book(String title) {
    this.title = title;
```

```
  }

  public synchronized void issue(int days) {
    dateIssued = Calendar.getInstance();
    dateDue = Calendar.getInstance();
    dateDue.add(dateIssued.DATE, days);
  }

  public synchronized Calendar getDueDate() {
    return dateDue;
  }
}
```

This class does not commit to its locking strategy (that is, it reserves the right to change its locking strategy without notice). Furthermore, it does not document that callers can use client-side locking safely. The `BookWrapper` client class uses client-side locking in the `renew()` method by synchronizing on a `Book` instance.

```
// Client
public class BookWrapper {
  private final Book book;

  BookWrapper(Book book) {
    this.book = book;
  }

  public void issue(int days) {
    book.issue(days);
  }

  public Calendar getDueDate() {
    return book.getDueDate();
  }

  public void renew() {
    synchronized(book) {
      if (book.getDueDate().before(Calendar.getInstance())) {
        throw new IllegalStateException("Book overdue");
      } else {
        book.issue(14); // Issue book for 14 days
      }
    }
  }
}
```

If the `Book` class changes its synchronization policy in the future, the `BookWrapper` class's locking strategy might silently break. For instance, the `BookWrapper` class's locking strategy breaks if `Book` is modified to use a private final lock object, as recommended by guideline "LCK00-J. Use private final lock objects to synchronize classes that may interact with untrusted

code" on page 41. The `BookWrapper` class's locking strategy breaks because threads that call `BookWrapper.getDueDate()` may perform operations on the thread-safe `Book` using its new locking policy. However, threads that call the `renew()`method will always synchronize on the intrinsic lock of the `Book` instance. Consequently, the implementation will use two different locks.

3.12.2 Compliant Solution (Private Final Lock Object)

This compliant solution uses a private final lock object and synchronizes the methods of the `BookWrapper` class using this lock.

```
public final class BookWrapper {
  private final Book book;
  private final Object lock = new Object();

  BookWrapper(Book book) {
    this.book = book;
  }

  public void issue(int days) {
    synchronized(lock) {
      book.issue(days);
    }
  }

  public Calendar getDueDate() {
    synchronized(lock) {
      return book.getDueDate();
    }
  }

  public void renew() {
    synchronized(lock) {
      if (book.getDueDate().before(Calendar.getInstance())) {
        throw new IllegalStateException("Book overdue");
      } else {
        book.issue(14); // Issue book for 14 days
      }
    }
  }
}
```

The `BookWrapper` class's locking strategy is now independent of the locking policy of the `Book` instance.

3.12.3 Noncompliant Code Example (Class Extension and Accessible Member Lock)

Goetz and colleagues describe the fragility of class extension for adding functionality to thread-safe classes [Goetz 2006]:

> *Extension is more fragile than adding code directly to a class, because the implementation of the synchronization policy is now distributed over multiple, separately maintained source files. If the underlying class were to change its synchronization policy by choosing a different lock to guard its state variables, the subclass would subtly and silently break, because it no longer used the right lock to control concurrent access to the base class's state.*

In this noncompliant code example, the `PrintableIPAddressList` class extends the thread-safe `IPAddressList` class. `PrintableIPAddressList` locks on `IPAddressList.ips` in the `addAndPrintIPAddresses()`method. This is another example of client-side locking because a subclass is using an object owned and locked by its superclass.

```
// This class may change its locking policy in the future, for example,
// if new non-atomic methods are added
class IPAddressList {
  private final List<InetAddress> ips =
    Collections.synchronizedList(new ArrayList<InetAddress>());

  public List<InetAddress> getList() {
    return ips; // No defensive copies required as package-private visibility
  }

  public void addIPAddress(InetAddress address) {
    ips.add(address);
  }
}

class PrintableIPAddressList extends IPAddressList {
  public void addAndPrintIPAddresses(InetAddress address) {
    synchronized(getList()) {
      addIPAddress(address);
      InetAddress[] ia = (InetAddress[]) getList().toArray(new InetAddress[0]);
      // ...
    }
  }
}
```

If the `IPAddressList` class is modified to use block synchronization on a private final lock object (as recommended by guideline "LCK00-J. Use private final lock objects to synchronize classes that may interact with untrusted code" on page 41), the `PrintableIPAddressList` subclass will silently break. Moreover, if a wrapper such as `Collections.synchronizedList()` is used, it is difficult for a client to determine the type of the class being wrapped in order to extend it [Goetz 2006].

3.12.4 Compliant Solution (Composition)

This compliant solution wraps an object of the `IPAddressList` class and provides synchronized accessors that can be used to manipulate the state of the object.

Composition offers encapsulation benefits, usually with minimal overhead. Refer to guideline "OBJ07-J. Understand how a superclass can affect a subclass for more information on composition."[8]

```
// Class IPAddressList remains unchanged
class PrintableIPAddressList {
  private final IPAddressList ips;

  public PrintableIPAddressList(IPAddressList list) {
    this.ips = list;
  }

  public synchronized void addIPAddress(InetAddress address) {
    ips.addIPAddress(address);
  }

  public synchronized void addAndPrintIPAddresses(InetAddress address) {
    addIPAddress(address);
    InetAddress[] ia = (InetAddress[]) ips.getList().toArray(new InetAddress[0]);
    // ...
  }
}
```

In this case, composition allows the `PrintableIPAddressList` class to use its own intrinsic lock independent of the underlying list class's lock. The underlying collection does not need to be thread-safe because the `PrintableIPAddressList` wrapper prevents direct access to its methods by publishing its own synchronized equivalents. This approach provides consistent locking even if the underlying class changes its locking policy in the future [Goetz 2006].

3.12.5 Risk Assessment

Using client-side locking when the thread-safe class does not commit to its locking strategy can cause data inconsistencies and deadlock.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
LCK11- J	low	probable	medium	P4	L3

3.12.6 References

[Goetz 2006]	Section 4.4.1, "Client-side Locking" Section 4.4.2, "Composition" Section 5.2.1, "ConcurrentHashMap"
[Lee 2009]	"Map & Compound Operation"
[Oaks 2004]	Section 8.2, "Synchronization and Collection Classes"
[Sun 2009b]	Class `Vector`, Class `WeakReference`, Class `ConcurrentHashMap<K,V>`

[8] This guideline is described at https://www.securecoding.cert.org/confluence/display/java/.

4 Thread APIs (THI) Guidelines

4.1 THI00-J. Do not assume that the sleep(), yield(), or getState() methods provide synchronization semantics

According to Section 17.9, "Sleep and Yield" of the *Java Language Specification* [Gosling 2005]

> *It is important to note that neither `Thread.sleep` nor `Thread.yield` have any synchronization semantics. In particular, the compiler does not have to flush writes cached in registers out to shared memory before a call to `Thread.sleep` or `Thread.yield`, nor does the compiler have to reload values cached in registers after a call to `Thread.sleep` or `Thread.yield`.*

Incorrectly assuming that thread suspension and yielding do any of the following can result in unexpected behavior:

- flush the cached registers
- reload any values
- provide any happens-before relationships when execution resumes

4.1.1 Noncompliant Code Example (`sleep()`)

This noncompliant code example attempts to use a non-volatile Boolean `done` as a flag to terminate the execution of a thread. A separate thread sets `done` to true by calling the `shutdown()` method.

```
final class ControlledStop implements Runnable {
  private boolean done = false;

  @Override public void run() {
    while (!done) {
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt(); // Reset interrupted status
      }
    }
  }

  public void shutdown() {
    this.done = true;
  }
}
```

However, the compiler is free to read the field `this.done` once and reuse the cached value in each execution of the loop. Consequently, the while loop might not terminate, even if another thread calls the `shutdown()` method to change the value of `this.done` [Gosling 2005]. This

error could have resulted from the programmer incorrectly assuming that the call to `Thread.sleep()` would cause cached values to be reloaded.

4.1.2 Compliant Solution (Volatile Flag)

This compliant solution declares the flag volatile to ensure that updates to it are made visible across multiple threads.

```
final class ControlledStop implements Runnable {
  private volatile boolean done = false;

  // ...
}
```

The volatile flag establishes a happens-before relationship between this thread and any other thread that sets `done`.

4.1.3 Compliant Solution (`Thread.interrupt()`)

A better solution for methods that call `sleep()` is to use thread interruption, which causes the sleeping thread to wake up immediately and handle the interruption.

```
final class ControlledStop implements Runnable {
  @Override public void run() {
    while (!Thread.interrupted()) {
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
      }
    }
  }

  public void shutdown() {
    Thread.currentThread().interrupt();
  }
}
```

4.1.4 Noncompliant Code Example (`getState()`)

This noncompliant code example starts a thread in the `doSomething()` method. The thread supports interruption by checking the volatile flag and blocks waiting until notified. The `stop()` method notifies the thread if it is blocked on the wait and sets the flag to true so that the thread can terminate.

```
public class Waiter {
  private Thread thread;
  private volatile boolean flag;
  private final Object lock = new Object();
```

```
  public void doSomething() {
    thread = new Thread(new Runnable() {
      @Override public void run() {
        synchronized(lock) {
          while (!flag) {
            try {
              lock.wait();
              // ...
            } catch (InterruptedException e) {
              // Forward to handler
            }
          }
        }
      }
    });
    thread.start();
  }

  public boolean stop() {
    if (thread != null) {
      if (thread.getState() == Thread.State.WAITING) {
        flag = true;
        synchronized (lock) {
          lock.notifyAll();
        }
        return true;
      }
    }
    return false;
  }
}
```

Unfortunately, the `stop()` method incorrectly uses the `Thread.getState()` method to check whether the thread is blocked and has not terminated before delivering the notification. Using the `Thread.getState()` method for synchronization control such as checking whether a thread is blocked on a wait is inappropriate. This is true because a blocked thread is not always required to enter the `WAITING` or `TIMED_WAITING` state in cases where the JVM implements blocking using spin-waiting [Goetz 2006]. Because the thread may never enter the `WAITING` state, the `stop()` method may not terminate the thread.

4.1.5 Compliant Solution

This compliant solution removes the check for determining whether the thread is in the `WAITING` state. This check is unnecessary because invoking `notifyAll()` on a thread that is not blocked on a `wait()` invocation has no effect.

```
public class Waiter {
  // ...

  public boolean stop() {
    if (thread != null) {
      flag = true;
      synchronized (lock) {
        lock.notifyAll();
      }
      return true;
    }
    return false;
  }
}
```

4.1.6 Risk Assessment

Relying on the Thread class's `sleep()`, `yield()`, and `getState()` methods for synchronization control can cause unexpected behavior.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
THI00- J	low	probable	medium	P4	L3

4.1.7 References

[Gosling 2005] Section 17.9, "Sleep and Yield"

4.2 THI01-J. Do not invoke ThreadGroup methods

Each thread in Java is assigned to a thread group upon the thread's creation. These groups are implemented by the `java.lang.ThreadGroup` class. If the thread group name is not specified explicitly, the `main` default group is assigned by the JVM [Sun 2008a]. The convenience methods of the `ThreadGroup` class can be used to operate on all threads belonging to a thread group at once. For example, the `ThreadGroup.interrupt()` method interrupts all threads in the thread group. Thread groups also help reinforce layered security by confining threads into groups so that they do not interfere with threads in other groups [Oaks 2004].

Even though thread groups are useful for keeping threads organized, programmers seldom benefit from their use because many of the `ThreadGroup` class methods are deprecated (for example, `allowThreadSuspension()`, `resume()`, `stop()`, and `suspend()`). Furthermore, many non-deprecated methods are obsolete in that they offer little desirable functionality. Ironically, a few `ThreadGroup` methods are not even thread-safe [Bloch 2001].

The insecure yet non-deprecated methods include

- `ThreadGroup.activeCount()`
  According to the Java API, the `activeCount()` method [Sun 2009b]

  > *Returns an estimate of the number of active threads in this thread group*

  This method is often used as a precursor to thread enumeration. If a thread is not started, it continues to reside in the thread group and is considered to be active. Furthermore, the active count is affected by the presence of certain system threads [Sun 2009b]. Consequently, the *activeCount()* method may not reflect the actual number of running tasks in the thread group.
- `ThreadGroup.enumerate()`
  According to the Java API, `ThreadGroup` class documentation [Sun 2009b]

  > *[The `enumerate()` method] Copies into the specified array every active thread in this thread group and its subgroups. An application should use the `activeCount` method to get an estimate of how big the array should be. If the array is too short to hold all the threads, the extra threads are silently ignored.*

Using the `ThreadGroup` APIs to shut down threads also has pitfalls. Because the `stop()` method is deprecated, alternative ways are required to stop threads. According to the Java Programming Language [Arnold 2006]

> *One way is for the thread initiating the termination to join the other threads and so know when those threads have terminated. However, an application may have to maintain its own list of the threads it creates because simply inspecting the `ThreadGroup` may return library threads that do not terminate and for which join will not return.*

The `Executor` framework provides a better API for managing a logical grouping of threads and offers secure facilities for handling shutdown and thread exceptions [Bloch 2008].

4.2.1 Noncompliant Code Example

This noncompliant code example contains a `NetworkHandler` class that maintains a `controller` thread. This thread delegates a new request to a worker thread. To demonstrate the race condition in this example, the `controller` thread services three requests by starting three

threads in succession from its `run()` method. All threads are defined to belong to the `Chief` thread group.

```
final class HandleRequest implements Runnable {
  public void run() {
    // Do something
  }
}

public final class NetworkHandler implements Runnable {
  private static ThreadGroup tg = new ThreadGroup("Chief");

  @Override public void run() {
    new Thread(tg, new HandleRequest(), "thread1").start(); // Start thread 1
    new Thread(tg, new HandleRequest(), "thread2").start(); // Start thread 2
    new Thread(tg, new HandleRequest(), "thread3").start(); // Start thread 3
  }

  public static void printActiveCount(int point) {
    System.out.println("Active Threads in Thread Group " + tg.getName() +
      " at point(" + point + "):" + " " + tg.activeCount());
  }

  public static void printEnumeratedThreads(Thread[] ta, int len) {
    System.out.println("Enumerating all threads...");
    for(int i = 0; i < len; i++) {
      System.out.println("Thread " + i + " = " + ta[i].getName());
    }
  }

  public static void main(String[] args) throws InterruptedException {
    // Start thread controller
    Thread thread = new Thread(tg, new NetworkHandler(), "controller");
    thread.start();

    Thread[] ta = new Thread[tg.activeCount()]; // Gets the active count (insecure)

    printActiveCount(1);  // P1
    Thread.sleep(1000);  // Delay to demonstrate TOCTOU condition (race window)
    printActiveCount(2);  // P2: the thread count changes as new threads are initiated
    // Incorrectly uses the (now stale) thread count obtained at P1
    int n = tg.enumerate(ta);
    printEnumeratedThreads(ta, n); // Silently ignores newly initiated threads
                                   // (between P1 and P2)

    // This code destroys the thread group if it does not have any alive threads
    for (Thread thr : ta) {
      thr.interrupt();
      while(thr.isAlive());
```

```
    }
    tg.destroy();
  }
}
```

There is a time-of-check-to-time-of-use (TOCTOU) vulnerability in this implementation because obtaining the count and enumerating the list do not constitute an atomic operation. If new requests occur after the call to `activeCount()` and before the call to `enumerate()` in the `main()` method, the total number of threads in the group will increase but the enumerated list `ta` will contain only the initial number, that is, two thread references (`main` and `controller`). Consequently, the program will fail to account for the newly started threads in the `Chief` thread group.

Any subsequent use of the `ta` array is insecure. For example, calling the `destroy()` method to destroy the thread group and its subgroups will not work as expected. The precondition to calling `destroy()` is that the thread group is empty with no executing threads. The code attempts to accomplish this by interrupting every thread in the thread group. However, when the `destroy()` method is called, the thread group is not empty, which causes a `java.lang.IllegalThreadStateException` to be thrown.

4.2.2 Compliant Solution

This compliant solution uses a fixed thread pool, rather than a `ThreadGroup`, to group its three tasks. The `java.util.concurrent.ExecutorService` interface provides methods to manage the thread pool. Note that there are no methods for finding the number of actively executing threads or for enumerating through them. However, the logical grouping can help control the behavior of the group as a whole. For instance, all threads belonging to a particular thread pool can be terminated by calling the `shutdownPool()` method.

```
public final class NetworkHandler {
  private final ExecutorService executor;

  NetworkHandler(int poolSize) {
    this.executor = Executors.newFixedThreadPool(poolSize);
  }

  public void startThreads() {
    for(int i = 0; i < 3; i++) {
      executor.execute(new HandleRequest());
    }
  }

  public void shutdownPool() {
    executor.shutdown();
  }
```

```
  public static void main(String[] args)  {
    NetworkHandler nh = new NetworkHandler(3);
    nh.startThreads();
    nh.shutdownPool();
  }
}
```

Before Java SE 5.0, the `ThreadGroup` class had to be extended because there was no other direct way to catch an uncaught exception in a separate thread. If the application had installed an `UncaughtExceptionHandler`, it could only be controlled by subclassing `ThreadGroup`. In recent versions, `UncaughtExceptionHandler` is maintained on a per-thread basis using an interface enclosed by the `Thread` class, which leaves little to no functionality for the `ThreadGroup` class [Goetz 2006, Bloch 2008].

Refer to guideline "TPS03-J. Ensure that tasks executing in a thread pool do not fail silently" on page 135 for more information on using uncaught exception handlers in thread pools.

4.2.3 Risk Assessment

Using the `ThreadGroup` APIs may result in race conditions, memory leaks, and inconsistent object state.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
THI01- J	low	probable	medium	P4	L3

4.2.4 References

[Arnold 2006] Section 23.3.3, "Shutdown Strategies"

[Bloch 2001] "Item 53: Avoid thread groups"

[Bloch 2008] "Item 73: Avoid thread groups"

[Goetz 2006] Section 7.3.1, "Uncaught Exception Handlers"

[Oaks 2004] Section 13.1, "ThreadGroups"

[Sun 2009b] Methods `activeCount` and `enumerate`, Classes `ThreadGroup` and `Thread`

[Sun 2008a]

[Sun 2008b] Bug ID: 4089701 and 4229558

4.3 THI02-J. Do not invoke Thread.run()

It is critical to ensure that threads are started correctly. Thread start-up can be misleading because sometimes the code appears to be performing the function correctly, when it is actually executing in the wrong thread.

The `Thread.start()` method starts executing a thread's `run()` method in the respective thread. It is a mistake to directly invoke the `run()` method on a `Thread` object. When invoked directly, the statements in the `run()` method execute in the current thread instead of the newly created thread. Furthermore, if the `Thread` object is not constructed from a `Runnable` object but rather by instantiating a subclass of `Thread` that does not override the `run()` method, a call to the subclass's `run()` method invokes `Thread.run()`, which does nothing.

4.3.1 Noncompliant Code Example

This noncompliant code example explicitly invokes the `run()` method in the context of the current thread.

```
public final class Foo implements Runnable {
  @Override public void run() {
    // ...
  }

  public static void main(String[] args) {
    Foo foo = new Foo();
    new Thread(foo).run();
  }
}
```

The `start()` method is not invoked on the new thread because of the incorrect assumption that `run()` starts the new thread. Consequently, the statements in the `run()` method execute in the same thread instead of the new one.

4.3.2 Compliant Solution

This compliant solution correctly uses the `start()` method to start a new thread. Then, that method internally invokes the `run()` method in the new thread.

```
public final class Foo implements Runnable {
  @Override public void run() {
    // ...
  }

  public static void main(String[] args) {
    Foo foo = new Foo();
    new Thread(foo).start();
  }
}
```

4.3.3 Exceptions

THI02-EX1: The `run()` method may be invoked when unit testing functionality. Note that this method cannot be used to test a class for multithreaded use.

Given a `Thread` object that has been constructed with a runnable argument, when invoking the `Thread.run()`method, the `Thread` object may be cast to `Runnable` to eliminate analyzer diagnostics.

```
Thread thread = new Thread(new Runnable() {
  @Override public void run() {
    // ...
  }
});

((Runnable) thread).run();  // Exception: This does not start a new thread
```

Casting a thread to `Runnable` before calling the `run()` method documents that the explicit call to `Thread.run()` is intentional. Adding an explanatory comment alongside the invocation is highly recommended.

4.3.4 Risk Assessment

Failing to start threads correctly can cause unexpected behavior.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
THI02- J	low	probable	medium	P4	L3

4.3.5 References

[Sun 2009b] Interface Runnable and class `Thread`

4.4 THI03-J. Always invoke wait() and await() methods inside a loop

The `Object.wait()` method temporarily cedes possession of a lock so that another thread that is requesting the lock can proceed. `Object.wait()` must always be called from a synchronized block or method. To resume the waiting thread, the requesting thread must invoke the `notify()` method to notify it. Furthermore, the `wait()` method should be invoked in a loop that checks if a condition predicate holds. Note that a condition predicate is the negation of the condition expression in the loop. For example, the condition predicate for removing an element from a vector is `!isEmpty()`, whereas the condition expression for the while loop condition is `isEmpty()`. The correct way to invoke the `wait()` method when the vector is empty is shown below.

```
public void consumeElement() throws InterruptedException {
  synchronized (vector) {
    while (vector.isEmpty()) {
      vector.wait();
    }

    // Consume when condition holds
  }
}
```

The notification mechanism notifies the waiting thread and lets it check its condition predicate. The invocation of the `notify()` or `notifyAll()` methods in another thread cannot precisely determine which waiting thread is resumed. A condition predicate statement is provided so that only the correct thread will resume upon receiving the notification. A condition predicate also helps when a thread is required to block until a condition becomes true, such as reading data from an input stream before proceeding.

Safety and liveness are both concerns when using the wait/notify mechanism. Safety requires all objects to maintain consistent states in a multithreaded environment [Lea 2000a]. Liveness requires that every operation or method invocation execute to completion without interruption.

To guarantee liveness, the `while` loop condition must be tested before the `wait()` method is invoked. This is done in case another thread has already satisfied the condition predicate and sent a notification. Invoking the `wait()` method after the notification has been sent results in indefinite blocking.

To guarantee safety, the `while` loop condition must be tested even after the `wait()` method is invoked. While `wait()` is meant to block indefinitely until a notification is received, it must still be encased within a loop to prevent the following vulnerabilities [Bloch 2001]:

- thread in the middle - A third thread can acquire the lock on the shared object during the interval between a notification being sent and the receiving thread resuming execution. This thread can change the state of the object, leaving it inconsistent. This is a time-of-check-to-time-of-use (TOCTOU) condition.
- malicious notification - There is no guarantee that a random notification will not be received when the condition predicate is false. This means that the invocation of `wait()` may be nullified by the notification.

- misdelivered notification - Sometimes on receipt of a `notifyAll()` signal, an unrelated thread can start executing, and it is possible for its condition predicate to be true. Consequently, it may resume execution although it was required to remain dormant.
- spurious wake-ups - Certain JVM implementations are vulnerable to spurious wake-ups that result in waiting threads waking up even without a notification [Sun 2009b].

For these reasons, the condition predicate must be checked after the `wait()` method is invoked. A while loop is the best choice for checking the condition predicate before and after invoking `wait()`.

Similarly, the `await()` method of the `Condition` interface must also be invoked inside a loop. According to the Java API [Sun 2009b], Interface `Condition`

> *When waiting upon a Condition, a "spurious wakeup" is permitted to occur, in general, as a concession to the underlying platform semantics. This has little practical impact on most application programs as a Condition should always be waited upon in a loop, testing the state predicate that is being waited for. An implementation is free to remove the possibility of spurious wakeups but it is recommended that applications programmers always assume that they can occur and so always wait in a loop.*

New code should use the `java.util.concurrent` concurrency utilities instead of the wait/notify mechanism. However, legacy code may depend on the wait/notify mechanism.

4.4.1 Noncompliant Code Example

This noncompliant code example invokes the `wait()` method inside a traditional `if` block and fails to check the post-condition after the notification is received. If the notification is accidental or malicious, the thread can wake up prematurely.

```
synchronized (object) {
  if (<condition does not hold>) {
    object.wait();
  }
  // Proceed when condition holds
}
```

4.4.2 Compliant Solution

This compliant solution calls the `wait()` method from within a `while` loop to check the condition before and after `wait()` is called.

```
synchronized (object) {
  while (<condition does not hold>) {
    object.wait();
  }
  // Proceed when condition holds
}
```

Similarly, invocations of the `await()` method of the `java.util.concurrent.locks.Condition` interface must be enclosed in a loop.

4.4.3 Risk Assessment

To guarantee liveness and safety, the `wait()` and `await()` methods must always be invoked inside a `while` loop.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
THI03- J	low	unlikely	medium	P2	L3

4.4.4 References

[Bloch 2001] Item 50: “Never invoke wait outside a loop”

[Goetz 2006] Section 14.2, “Using Condition Queues”

[Lea 2000a] Section 3.2.2, “Monitor Mechanics”
Section 1.3.2, “Liveness”

[Sun 2009b] Class Object

4.5 THI04-J. Notify all waiting threads instead of a single thread

A thread that invokes `wait()` expects to wake up and resume execution when its condition predicate becomes true. Waiting threads must test their condition predicates upon receiving notifications and resume waiting if the predicates are false, to be compliant with guideline “THI03-J. Always invoke wait() and await() methods inside a loop” on page 101.

The `notify()` and `notifyAll()` methods of the `java.lang.Object` package are used to wake up waiting thread(s). These methods must be invoked from code that holds the same object lock as the waiting thread(s). An `IllegalMonitorStateException` is thrown if the current thread does not acquire this object’s intrinsic lock before invoking these methods. The `notifyAll()` method wakes up all threads and allows threads whose condition predicate is true to resume execution. Furthermore, if all the threads whose condition predicate evaluates to true previously held a specific lock before going into the wait state, only one of them will reacquire the lock upon being notified. Presumably, the other threads will resume waiting. The `notify()` method wakes up only one thread and makes no guarantees as to which thread is notified. If the thread’s condition predicate doesn’t allow the thread to proceed, the chosen thread may resume waiting, defeating the purpose of the notification.

The `notify()` method may only be invoked if all of the following conditions are met:

- The condition predicate is identical for each waiting thread.
- All threads must perform the same set of operations after waking up. This means that any one thread can be selected to wake up and resume for a single invocation of `notify()`.
- Only one thread is required to wake upon the notification.

These conditions are satisfied by threads that are identical and provide a stateless service or utility.

The `java.util.concurrent` utilities (`Condition` interface) provide the `signal()` and `signalAll()` methods to awaken threads that are blocked on an `await()` call. `Condition` objects are required when using `Lock` objects. A `Lock` object allows the use of the `wait()` and `notify()` methods. However, code that synchronizes using a `Lock` object does not use its own intrinsic lock. Instead, one or more `Condition` objects are associated with the `Lock` object. These objects interact directly with the locking policy enforced by the `Lock` object. Consequently, the `Condition.await()`, `Condition.signal()`, and `Condition.signalAll()` methods are used instead of `Object.wait()`, `Object.notify()`, and `Object.notifyAll()`.

The use of the `signal()` method is insecure when multiple threads await the same `Condition` object unless all of the following conditions are met:

- The `Condition` object is identical for each waiting thread.
- All threads must perform the same set of operations after waking up. This means that any one thread can be selected to wake up and resume for a single invocation of `signal()`.
- Only one thread is required to wake upon receiving the signal.

The `signal()` method may also be invoked when both of the following conditions are met:

- Each thread uses a unique `Condition` object.

- Each `Condition` object is associated with a common `Lock` object.

The `signal()` method, if used securely, has better performance than `signalAll()`.

4.5.1 Noncompliant Code Example (`notify()`)

This noncompliant code example shows a complex multistep process being undertaken by several threads. Each thread executes the step identified by the `time` field. Each thread waits for the `time` field to indicate that it is time to perform the corresponding thread's step. After performing the step, each thread increments `time` and then notifies the thread that is responsible for performing the next step.

```
public final class ProcessStep implements Runnable {
  private static final Object lock = new Object();
  private static int time = 0;
  private final int step; // Do operations when field time reaches this value

  public ProcessStep(int step) {
    this.step = step;
  }

  @Override public void run() {
    try {
      synchronized (lock) {
        while (time != step) {
          lock.wait();
        }

        // Perform operations

        time++;
        lock.notify();
      }
    } catch (InterruptedException ie) {
      Thread.currentThread().interrupt(); // Reset interrupted status
    }
  }

  public static void main(String[] args) {
    for (int i = 4; i >= 0; i--) {
      new Thread(new ProcessStep(i)).start();
    }
  }
}
```

This noncompliant code example violates the liveness property. Each thread has a different condition predicate, as each requires `step` to have a different value before proceeding. The `Object.notify()` method wakes up only one thread at a time. Unless it happens to wake up the thread that is required to perform the next step, the program will deadlock.

4.5.2 Compliant Solution (`notifyAll()`)

In this compliant solution, each thread completes its step and then calls `notifyAll()` to notify the waiting threads. The thread that is ready can then perform its task, while all the threads whose condition predicates are false (loop condition expression is true) promptly resume waiting.

Only the `run()` method from the noncompliant code example is modified, as follows:

```
@Override public void run() {
  try {
    synchronized (lock) {
      while (time != step) {
        lock.wait();
      }

      // Perform operations

      time++;
      lock.notifyAll(); // Use notifyAll() instead of notify()
    }
  } catch (InterruptedException ie) {
    Thread.currentThread().interrupt(); // Reset interrupted status
  }
}
```

4.5.3 Noncompliant Code Example (`Condition` interface)

This noncompliant code example is similar to the noncompliant code example for `notify()` but uses the `Condition` interface for waiting and notification.

```
public class ProcessStep implements Runnable {
  private static final Lock lock = new ReentrantLock();
  private static final Condition condition = lock.newCondition();
  private static int time = 0;
  private final int step; // Do operations when field time reaches this value

  public ProcessStep(int step) {
    this.step = step;
  }

  @Override public void run() {
    lock.lock();
    try {
      while (time != step) {
        condition.await();
      }

      // Perform operations
```

```
      time++;
      condition.signal();
    } catch (InterruptedException ie) {
      Thread.currentThread().interrupt(); // Reset interrupted status
    } finally {
      lock.unlock();
    }
  }

  public static void main(String[] args) {
    for (int i = 4; i >= 0; i--) {
      new Thread(new ProcessStep(i)).start();
    }
  }
}
```

As with `Object.notify()`, the `signal()` method may awaken an arbitrary thread.

4.5.4 Compliant Solution (`signalAll()`)

This compliant solution uses the `signalAll()` method to notify all waiting threads. Before `await()` returns, the current thread reacquires the lock associated with this condition. When the thread returns, it is guaranteed to hold this lock [Sun 2009b]. The thread that is ready can perform its task, while all the threads whose condition predicates are false resume waiting.

Only the `run()` method from the noncompliant code example is modified, as follows:

```
  @Override public void run() {
    lock.lock();
    try {
      while (time != step) {
        condition.await();
      }

      // Perform operations

      time++;
      condition.signalAll();
    } catch (InterruptedException ie) {
      Thread.currentThread().interrupt(); // Reset interrupted status
    } finally {
      lock.unlock();
    }
  }
```

4.5.5 Compliant Solution (Unique Condition Per Thread)

This compliant solution assigns each thread its own condition. All the `Condition` objects are accessible to all the threads.

```
// Declare class as final because its constructor throws an exception
public final class ProcessStep implements Runnable {
  private static final Lock lock = new ReentrantLock();
  private static int time = 0;
  private final int step; // Do operations when field time reaches this value
  private static final int MAX_STEPS = 5;
  private static final Condition[] conditions = new Condition[MAX_STEPS];

  public ProcessStep(int step) {
    if (step <= MAX_STEPS) {
      this.step = step;
      conditions[step] = lock.newCondition();
    } else {
      throw new IllegalArgumentException("Too many threads");
    }
  }

  @Override public void run() {
    lock.lock();
    try {
      while (time != step) {
        conditions[step].await();
      }

      // Perform operations

      time++;
      if (step + 1 < conditions.length) {
        conditions[step + 1].signal();
      }
    } catch (InterruptedException ie) {
      Thread.currentThread().interrupt(); // Reset interrupted status
    } finally {
      lock.unlock();
    }
  }

  public static void main(String[] args) {
    for (int i = MAX_STEPS - 1; i >= 0; i--) {
      ProcessStep ps = new ProcessStep(i);
      new Thread(ps).start();
    }
  }
}
```

Even though the `signal()` method is used, only the thread whose condition predicate corresponds to the unique `Condition` variable will awaken.

This compliant solution is safe only if untrusted code cannot create a thread with an instance of this class.

4.5.6 Risk Assessment

Notifying a single thread instead of all waiting threads can pose a threat to the liveness property of the system.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
THI04- J	low	unlikely	medium	P2	L3

4.5.7 References

[Bloch 2001] Item 50: “Never invoke wait outside a loop”

[Goetz 2006] Section 14.2.4, “Notification”

[Gosling 2005] Chapter 17, “Threads and Locks”

[Sun 2009b] `java.util.concurrent.locks.Condition` interface

4.6 THI05-J. Do not use Thread.stop() to terminate threads

Threads always preserve class invariants when they are allowed to exit normally. Programmers often try to terminate threads abruptly when they believe that the task is accomplished, the request has been canceled, or the program or the JVM needs to shut down quickly.

A few thread APIs were introduced to facilitate thread suspension, resumption, and termination but were later deprecated because of inherent design weaknesses. For example, the `Thread.stop()` method causes the thread to immediately throw a `ThreadDeath` exception, which usually stops the thread.

Invoking `Thread.stop()` results in the release of all the locks a thread has acquired, which may corrupt the state of the object. The thread could catch the `ThreadDeath` exception and use a `finally` block in an attempt to repair the inconsistent object. However, that requires careful inspection of all the synchronized methods and blocks because a `ThreadDeath` exception can be thrown at any point during the thread's execution. Furthermore, code must be protected from `ThreadDeath` exceptions that may result when executing `catch` or `finally` blocks [Sun 1999a].

More information about deprecated methods is available in guideline "MET15-J. Do not use deprecated or obsolete methods."[9] Also, refer to guideline "EXC09-J. Prevent inadvertent calls to `System.exit()` or forced shutdown"[8] for information on preventing data corruption when the JVM is shut down abruptly.

4.6.1 Noncompliant Code Example (Deprecated `Thread.stop()`)

This noncompliant code example shows a thread that fills a vector with pseudo-random numbers. The thread is forcefully stopped after a given amount of time.

```
public final class Container implements Runnable {
  private final Vector<Integer> vector = new Vector<Integer>(1000);

  public Vector<Integer> getVector() {
    return vector;
  }

  @Override public synchronized void run() {
    Random number = new Random(123L);
    int i = vector.capacity();
    while (i > 0) {
      vector.add(number.nextInt(100));
      i--;
    }
  }

  public static void main(String[] args) throws InterruptedException {
    Thread thread = new Thread(new Container());
```

[9] This guideline is described at https://www.securecoding.cert.org/confluence/display/java/.

```
    thread.start();
    Thread.sleep(5000);
    thread.stop();
  }
}
```

Because the `Vector` class is thread-safe, operations performed by multiple threads on its shared instance are expected to leave it in a consistent state. For instance, the `Vector.size()` method always returns the correct number of elements in the vector even in the face of concurrent changes to the vector. This is because the vector instance uses its own intrinsic lock to prevent other threads from accessing it while its state is temporarily inconsistent.

However, the `Thread.stop()` method causes the thread to stop what it is doing and throw a `ThreadDeath` exception. All acquired locks are subsequently released [Sun 2009b]. If the thread is in the process of adding a new integer to the vector when it is stopped, the vector may become accessible while it is in an inconsistent state. This can result in `Vector.size()` returning an incorrect element count, for example, because the element count is incremented after adding the element.

4.6.2 Compliant Solution (Volatile Flag)

This compliant solution uses a volatile flag to terminate the thread. The `shutdown()`accessor method is used to set the flag to true. The thread's `run()` method polls the `done` flag and terminates when it becomes true.

```
public final class Container implements Runnable {
  private final Vector<Integer> vector = new Vector<Integer>(1000);
  private volatile boolean done = false;

  public Vector<Integer> getVector() {
    return vector;
  }

  public void shutdown() {
    done = true;
  }

  @Override public synchronized void run() {
    Random number = new Random(123L);
    int i = vector.capacity();
    while (!done && i > 0) {
      vector.add(number.nextInt(100));
      i--;
    }
  }

  public static void main(String[] args) throws InterruptedException {
    Container container = new Container();
    Thread thread = new Thread(container);
```

```
    thread.start();
    Thread.sleep(5000);
    container.shutdown();
  }
}
```

4.6.3 Compliant Solution (Interruptible)

In this compliant solution, the `Thread.interrupt()` method is called from `main()` to terminate the thread. Invoking `Thread.interrupt()` sets an internal interrupt status flag. The thread polls that flag using the `Thread.interrupted()` method, which returns true if the current thread has been interrupted and clears the interrupt status.

```
public final class Container implements Runnable {
  private final Vector<Integer> vector = new Vector<Integer>(1000);

  public Vector<Integer> getVector() {
    return vector;
  }

  @Override public synchronized void run() {
    Random number = new Random(123L);
    int i = vector.capacity();
    while (!Thread.interrupted() && i > 0) {
      vector.add(number.nextInt(100));
      i--;
    }
  }

  public static void main(String[] args) throws InterruptedException {
    Container c = new Container();
    Thread thread = new Thread(c);
    thread.start();
    Thread.sleep(5000);
    thread.interrupt();
  }
}
```

A thread may use interruption for performing tasks other than cancellation and shutdown. Consequently, a thread should not be interrupted unless its interruption policy is known in advance. Failure to do so can result in failed interruption requests.

4.6.4 Compliant Solution (Runtime Permission `stopThread`)

Removing the default `java.lang.RuntimePermission stopThread` permission from the security policy file prevents threads from being stopped using the `Thread.stop()` method. This approach is not recommended for trusted, custom-developed code that uses that method because the existing design presumably depends on the ability of the system to perform this action. Furthermore, the system may not be designed to properly handle the resulting exception. In these cases, it is preferable to implement an alternate design approach corresponding to another compliant solution described in this guideline.

4.6.5 Risk Assessment

Forcing a thread to stop can result in inconsistent object state. Critical resources may also leak if clean-up operations are not carried out as required.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
THI05- J	low	probable	medium	P4	L3

4.6.6 References

[Arnold 2006] Section 14.12.1, “Don’t stop”
Section 23.3.3, “Shutdown Strategies”

[Darwin 2004] Section 24.3, “Stopping a Thread”

[Goetz 2006] Chapter 7, “Cancellation and shutdown”

[Oaks 2004] Section 2.4, “Two Approaches to Stopping a Thread”

[Sun 2009b] Class Thread, method `stop`, interface ExecutorService

[Sun 2008c] Concurrency Utilities, More information: Java Thread Primitive Deprecation

[Sun 99]

4.7 THI06-J. Ensure that threads and tasks performing blocking operations can be terminated

Threads and tasks that block on operations involving network or file input/output (I/O) must provide callers with an explicit termination mechanism to prevent denial-of-service vulnerabilities.

4.7.1 Noncompliant Code Example (Blocking I/O, Volatile Flag)

This noncompliant code example uses a volatile `done` flag to indicate that it is safe to shut down the thread, as suggested in guideline "THI05-J. Do not use `Thread.stop()` to terminate threads" on page 110. However, setting the flag does not terminate the thread if it is blocked on network I/O as a consequence of invoking the `readLine()` method.

```
public final class SocketReader implements Runnable { // Thread-safe class
  private final Socket socket;
  private final BufferedReader in;
  private volatile boolean done = false;
  private final Object lock = new Object();

  public SocketReader(String host, int port) throws IOException {
    this.socket = new Socket(host, port);
    this.in = new BufferedReader(new InputStreamReader(this.socket.getInputStream()));
  }

  // Only one thread can use the socket at a particular time
  @Override public void run() {
    try {
      synchronized (lock) {
        readData();
      }
    } catch (IOException ie) {
      // Forward to handler
    }
  }

  public void readData() throws IOException {
    String string;
    while (!done && (string = in.readLine()) != null) {
      // Blocks until end of stream (null)
    }
  }

  public void shutdown() {
    done = true;
  }

  public static void main(String[] args) throws IOException, InterruptedException {
    SocketReader reader = new SocketReader("somehost", 25);
    Thread thread = new Thread(reader);
```

```
    thread.start();
    Thread.sleep(1000);
    reader.shutdown(); // Shutdown the thread
  }
}
```

4.7.2 Noncompliant Code Example (Blocking I/O, Interruptible)

This noncompliant code example is similar to the preceding one but uses thread interruption to shut down the thread. Network I/O is not responsive to thread interruption when a `java.net.Socket` is being used. The `readData()` and `main()` methods are modified as follows:

```
public final class SocketReader implements Runnable { // Thread-safe class
  // ...

  public void readData() throws IOException {
    String string;
    while (!Thread.interrupted() && (string = in.readLine()) != null) {
      // Blocks until end of stream (null)
    }
  }

  public static void main(String[] args) throws IOException, InterruptedException {
    SocketReader reader = new SocketReader("somehost", 25);
    Thread thread = new Thread(reader);
    thread.start();
    Thread.sleep(1000);
    thread.interrupt(); // Interrupt the thread
  }
}
```

4.7.3 Compliant Solution (Close Socket Connection)

This compliant solution resumes the thread by having the `shutdown()` method close the socket. The `readLine()` method throws a `SocketException` when the socket is closed, which lets the thread proceed. Note that there is no way to keep the connection alive if the thread is to be halted cleanly and immediately.

```
public final class SocketReader implements Runnable {
  // ...

  public void readData() throws IOException {
    String string;
    try {
      while ((string = in.readLine()) != null) {
        // Blocks until end of stream (null)
      }
    } finally {
```

```
      shutdown();
    }
  }

  public void shutdown() throws IOException {
    socket.close();
  }

  public static void main(String[] args) throws IOException, InterruptedException {
    SocketReader reader = new SocketReader("somehost", 25);
    Thread thread = new Thread(reader);
    thread.start();
    Thread.sleep(1000);
    reader.shutdown();
  }
}
```

After the `shutdown()` method is called from `main()`, the `finally` block in `readData()` executes and calls `shutdown()` again, closing the socket for a second time. However, this second call has no effect if the socket has already been closed.

When performing asynchronous I/O, a `java.nio.channels.Selector` may also be brought out of the blocked state by invoking either its `close()` or `wakeup()` method.

A `boolean` flag can be used if additional operations need to be performed after emerging from the blocked state. When supplementing the code with such a flag, the `shutdown()` method should also set the flag to false so that the thread can exit cleanly from the while loop.

4.7.4 Compliant Solution (Interruptible Channel)

This compliant solution uses an interruptible channel, `java.nio.channels.SocketChannel`, instead of a `Socket` connection. If the thread performing the network I/O is interrupted using the `Thread.interrupt()` method while it is reading the data, the thread receives a `ClosedByInterruptException`, and the channel is closed immediately. The thread's interrupted status is also set.

```
public final class SocketReader implements Runnable {
  private final SocketChannel sc;
  private final Object lock = new Object();

  public SocketReader(String host, int port) throws IOException {
    sc = SocketChannel.open(new InetSocketAddress(host, port));
  }

  @Override public void run() {
    ByteBuffer buf = ByteBuffer.allocate(1024);
    try {
      synchronized (lock) {
        while (!Thread.interrupted()) {
          sc.read(buf);
```

```
          // ...
        }
      }
    } catch (IOException ie) {
      // Forward to handler
    }
  }

  public static void main(String[] args) throws IOException, InterruptedException {
    SocketReader reader = new SocketReader("somehost", 25);
    Thread thread = new Thread(reader);
    thread.start();
    Thread.sleep(1000);
    thread.interrupt();
  }
}
```

This technique interrupts the current thread. However, it only stops the thread because the code polls the thread's interrupted status with the `Thread.interrupted()` method and terminates the thread when it is interrupted. Using a `SocketChannel` ensures that the condition in the while loop is tested as soon as an interruption is received, despite the read being a blocking operation. Similarly, invoking the `interrupt()` method of a thread that is blocked because of `java.nio.channels.Selector` also causes that thread to awaken.

4.7.5 Noncompliant Code Example (Database Connection)

This noncompliant code example shows a thread-safe `DBConnector` class that creates one Java Database Connectivity (JDBC) connection per thread. Each connection belongs to one thread and is not shared by other threads. This is a common use case because JDBC connections are not meant to be shared by multiple threads.

```
public final class DBConnector implements Runnable {
  private final String query;

  DBConnector(String query) {
    this.query = query;
  }

  @Override public void run() {
    Connection connection;
    try {
      // Username and password are hard-coded for brevity
      connection = DriverManager.getConnection(
        "jdbc:driver:name",
        "username",
        "password"
      );
      Statement stmt = connection.createStatement();
      ResultSet rs = stmt.executeQuery(query);
```

```
      // ...
    } catch (SQLException e) {
      // Forward to handler
    }
    // ...
  }

  public static void main(String[] args) throws InterruptedException {
    DBConnector connector = new DBConnector("suitable query");
    Thread thread = new Thread(connector);
    thread.start();
    Thread.sleep(5000);
    thread.interrupt();
  }
}
```

Database connections, like sockets, are not inherently interruptible. Consequently, this design does not permit a client to cancel a task by closing the resource if the corresponding thread is blocked on a long-running query such as a join.

4.7.6 Compliant Solution (`Statement.cancel()`)

This compliant solution uses a `ThreadLocal` wrapper around the connection so that a thread calling the `initialValue()` method obtains a unique connection instance. The advantage of this approach is that a `cancelStatement()` method can be provided so that other threads or clients can interrupt a long-running query when required. The `cancelStatement()` method invokes the `Statement.cancel()`method.

```
public final class DBConnector implements Runnable {
  private final String query;
  private volatile Statement stmt;

  DBConnector(String query) {
    this.query = query;
    if (getConnection() != null) {
      try {
        stmt = getConnection().createStatement();
      } catch (SQLException e) {
        // Forward to handler
      }
    }
  }

  private static final ThreadLocal<Connection> connectionHolder =
    new ThreadLocal<Connection>() {
    Connection connection = null;

    @Override public Connection initialValue() {
      try {
```

```
      // ...
      connection = DriverManager.getConnection(
        "jdbc:driver:name",
        "username",
        "password"
      );
    } catch (SQLException e) {
      // Forward to handler
    }
    return connection;
  }
};

public Connection getConnection() {
  return connectionHolder.get();
}

public boolean cancelStatement() { // Allows client to cancel statement
  if (stmt != null) {
    try {
       stmt.cancel();
      return true;
    } catch (SQLException e) {
      // Forward to handler
    }
  }
  return false;
}

@Override public void run() {
  try {
    if(stmt == null || (stmt.getConnection() != getConnection())) {
      throw new IllegalStateException();
    }
    ResultSet rs = stmt.executeQuery(query);
    // ...
  } catch (SQLException e) {
    // Forward to handler
  }
  // ...
}
```

```
  public static void main(String[] args) throws InterruptedException {
    DBConnector connector = new DBConnector("suitable query");
    Thread thread = new Thread(connector);
    thread.start();
    Thread.sleep(5000);
    connector.cancelStatement();
  }
}
```

The `Statement.cancel()` method cancels the query, provided that the database management system (DBMS) and driver both support cancellation. It is not possible to conform with this guideline if they do not.

According to the Java API, interface `Statement` documentation [Sun 2009b]

> *By default, only one `ResultSet` object per `Statement` object can be open at the same time. Therefore, if the reading of one `ResultSet` object is interleaved with the reading of another, each must have been generated by different `Statement` objects.*

This compliant solution ensures that only one `ResultSet` is associated with the `Statement` belonging to an instance, and, consequently, only one thread can access the query results.

4.7.7 Risk Assessment

Failing to provide facilities for thread termination can cause non-responsiveness and denial of service.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
THI06- J	low	probable	medium	P4	L3

4.7.8 References

[Arnold 2006]	Section 14.12.1, “Don't stop” Section 23.3.3, “Shutdown Strategies”
[Darwin 2004]	Section 24.3, “Stopping a Thread”
[Goetz 2006]	Chapter 7, “Cancellation and shutdown”
[Oaks 2004]	Section 2.4, “Two Approaches to Stopping a Thread”
[Sun 2009b]	Class `Thread`, method `stop`, interface `ExecutorService`
[Sun 2008c]	Concurrency Utilities, More information: “Java Thread Primitive Deprecation”

5 Thread Pools (TPS) Guidelines

5.1 TPS00-J. Use thread pools to enable graceful degradation of service during traffic bursts

Many programs must address the problem of handling a series of incoming requests. The Thread-Per-Message design pattern is the simplest concurrency strategy wherein a new thread is created for each request [Lea 2000a]. This pattern is generally preferred to sequential executions of time-consuming, I/O-bound, session-based, or isolated tasks.

However, this pattern also has several pitfalls, including overheads of thread-creation and scheduling, task processing, resource allocation and deallocation, and frequent context switching [Lea 2000a]. Furthermore, an attacker can cause a denial of service by overwhelming the system with too many requests all at once. Instead of degrading gracefully, the system becomes unresponsive, causing a denial of service. From a safety perspective, one component can exhaust all resources because of some intermittent error, starving all other components.

Thread pools allow a system to service as many requests as it can comfortably sustain, rather than terminating all services when presented with a deluge of requests. Thread pools overcome these issues by controlling the maximum number of worker threads that can be initialized and executed concurrently. Every object that supports thread pools accepts a `Runnable` or `Callable<T>` task and stores it in a temporary queue until resources become available. Because the threads in a thread pool can be reused and efficiently added or removed from the pool, thread life-cycle management overhead is minimized.

5.1.1 Noncompliant Code Example

This noncompliant code example demonstrates the Thread-Per-Message design pattern. The `RequestHandler` class provides a public static factory method so that callers can obtain its instance. The `handleRequest()` method is subsequently invoked to handle each request in its own thread.

```
class Helper {
  public void handle(Socket socket) {
    //...
  }
}

final class RequestHandler {
  private final Helper helper = new Helper();
  private final ServerSocket server;

  private RequestHandler(int port) throws IOException {
    server = new ServerSocket(port);
  }
```

```
  public static RequestHandler newInstance() throws IOException {
    return new RequestHandler(0); // Selects next available port
  }

  public void handleRequest() {
    new Thread(new Runnable() {
      public void run() {
        try {
          helper.handle(server.accept());
        } catch (IOException e) {
          // Forward to handler
        }
      }
    }).start();
  }

}
```

The Thread-Per-Message strategy fails to provide graceful degradation of service. As more threads are created, processing continues normally until some scarce resource is exhausted. For example, a system may allow only a limited number of open file descriptors, even though several more threads can be created to service requests. When the scarce resource is memory, the system may fail abruptly, resulting in a denial of service.

5.1.2 Compliant Solution

This compliant solution uses a fixed-thread pool that places an upper bound on the number of concurrently executing threads. Tasks submitted to the pool are stored in an internal queue. That prevents the system from being overwhelmed when trying to respond to all the incoming requests and allows it to degrade gracefully by serving a fixed number of clients at a particular time [Sun 2008a].

```
// class Helper remains unchanged

final class RequestHandler {
  private final Helper helper = new Helper();
  private final ServerSocket server;
  private final ExecutorService exec;

  private RequestHandler(int port, int poolSize) throws IOException {
    server = new ServerSocket(port);
    exec = Executors.newFixedThreadPool(poolSize);
  }

  public static RequestHandler newInstance(int poolSize) throws IOException {
    return new RequestHandler(0, poolSize);
  }

  public void handleRequest() {
```

```
    Future<?> future = exec.submit(new Runnable() {
      @Override public void run() {
          try {
            helper.handle(server.accept());
          } catch (IOException e) {
            // Forward to handler
        }
      }
    });
  }
  // ... other methods such as shutting down the thread pool and task cancellation ...
}
```

According to the Java API documentation for the `Executor` interface [Sun 2009b]

> *[The Interface `Executor` is] An object that executes submitted `Runnable` tasks. This interface provides a way of decoupling task submission from the mechanics of how each task will be run, including details of thread use, scheduling, etc. An `Executor` is normally used instead of explicitly creating threads.*

The `ExecutorService` interface used in this compliant solution derives from the `java.util.concurrent.Executor` interface. The `ExecutorService.submit()` method allows callers to obtain a `Future<V>` object. This object encapsulates the as-yet-unknown result of an asynchronous computation and enables callers to perform additional functions such as task cancellation.

The choice of the unbounded `newFixedThreadPool` is not always optimal. Refer to the Java API documentation about choosing between the following to meet specific design requirements [Sun 2009b]:

- `newFixedThreadPool()`
- `newCachedThreadPool()`
- `newSingleThreadExecutor()`
- `newScheduledThreadPool()`

5.1.3 Risk Assessment

Using simplistic concurrency primitives to process an unbounded number of requests may result in severe performance degradation, deadlock, or system resource exhaustion and denial of service.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
TPS00- J	low	probable	high	**P2**	**L3**

5.1.4 References

[Goetz 2006] Chapter 8, "Applying Thread Pools"

[Lea 2000a] Section 4.1.3, "Thread-Per-Message"
Section 4.1.4, "Worker Threads"

[MITRE 2010] CWE ID 405, "Asymmetric Resource Consumption (Amplification)"
CWE ID 410, "Insufficient Resource Pool"

[Sun 2009b] Interface Executor

[Sun 2008a] Thread Pools

5.2 TPS01-J. Do not execute interdependent tasks in a bounded thread pool

A bounded thread pool allows the programmer to specify the upper limit on the number of threads that can execute in a thread pool at a particular time. Tasks that depend on the completion of other tasks should not be executed in a bounded thread pool.

A form of deadlock called *thread-starvation deadlock* arises when all the threads executing in the pool are blocked on tasks that have not yet begun executing and are waiting on an internal queue. Thread-starvation deadlock occurs when currently executing tasks submit other tasks to a thread pool and wait for them to complete, but the thread pool does not have the capacity to accommodate all the tasks at once.

This problem is deceptive because the program may appear to function correctly when fewer threads are needed. In some cases, the issue can be mitigated by choosing a larger pool size; however, there is often no easy way to determine a suitable size.

Similarly, threads in a thread pool may not be recycled if two executing tasks require each other to complete before they can terminate. A blocking operation within a subtask can also lead to unbounded, queue growth [Goetz 2006].

5.2.1 Noncompliant Code Example (Interdependent Subtasks)

This noncompliant code example is vulnerable to thread-starvation deadlock. It consists of the `ValidationService` class, which performs various input validation tasks such as checking whether a user-supplied field exists in a back-end database.

The `fieldAggregator()` method accepts a variable number of `String` arguments and creates a task corresponding to each argument to parallelize processing. The task performs input validation using the `ValidateInput` class.

In turn, the `ValidateInput` class attempts to sanitize the input by creating a subtask for each request using the `SanitizeInput` class. All tasks are executed in the same thread pool. The `fieldAggregator()` method blocks until all the tasks have finished executing and, when all results are available, returns the aggregated results as a `StringBuilder` object to the caller.

```
public final class ValidationService {
  private final ExecutorService pool;

  public ValidationService(int poolSize) {
    pool = Executors.newFixedThreadPool(poolSize);
  }

  public void shutdown() {
    pool.shutdown();
  }

  public StringBuilder fieldAggregator(String... inputs)
    throws InterruptedException, ExecutionException {
```

```
    StringBuilder sb = new StringBuilder();
    Future<String>[] results = new Future[inputs.length]; // Stores the results

    for (int i = 0; i < inputs.length; i++) { // Submits the tasks to thread pool
      results[i] = pool.submit(new ValidateInput<String>(inputs[i], pool));
    }

    for (int i = 0; i < inputs.length; i++) { // Aggregates the results
      sb.append(results[i].get());
    }
    return sb;
  }
}

public final class ValidateInput<V> implements Callable<V> {
  private final V input;
  private final ExecutorService pool;

  ValidateInput(V input, ExecutorService pool) {
    this.input = input;
    this.pool = pool;
  }

  @Override public V call() throws Exception {
    // If validation fails, throw an exception here
    Future<V> future = pool.submit(new SanitizeInput<V>(input)); // Subtask
    return (V)future.get();
  }
}

public final class SanitizeInput<V> implements Callable<V> {
  private final V input;

  SanitizeInput(V input) {
    this.input = input;
  }

  @Override public V call() throws Exception {
    // Sanitize input and return
    return (V)input;
  }
}
```

Assuming that the pool size is set to six, the `ValidationService.fieldAggregator()` method is invoked to validate the six arguments and submit six tasks to the thread pool. Each task submits corresponding subtasks to sanitize the input. The `SanitizeInput` subtasks must execute before these threads can return their results. However, this is impossible because all six threads in the thread pool are blocked. Furthermore, the `shutdown()` method cannot shut down the thread pool when it contains active tasks.

Thread-starvation deadlock can also occur when a single threaded `Executor` is used, for example, when the caller creates several subtasks and waits for the results.

5.2.2 Compliant Solution (No Interdependent Tasks)

This compliant solution modifies the `ValidateInput<V>` class so that the `SanitizeInput` tasks are executed in the same threads as the `ValidateInput` tasks and not in separate threads. Consequently, the `ValidateInput` and `SanitizeInput` tasks are independent and need not wait for each other to complete. The `SanitizeInput` class has also been modified to not implement the `Callable` interface.

```
public final class ValidationService {
  // ...
  public StringBuilder fieldAggregator(String... inputs)
      throws InterruptedException, ExecutionException {
    // ...
    for (int i = 0; i < inputs.length; i++) {
      // Don't pass-in thread pool
      results[i] = pool.submit(new ValidateInput<String>(inputs[i]));
    }
    // ...
  }
}

// Does not use same thread pool
public final class ValidateInput<V> implements Callable<V> {
  private final V input;

  ValidateInput(V input) {
    this.input = input;
  }

  @Override public V call() throws Exception {
    // If validation fails, throw an exception here
    return (V) new SanitizeInput().sanitize(input);
  }
}

public final class SanitizeInput<V> {  // No longer a Callable task
  public SanitizeInput() {}

  public V sanitize(V input) {
    // Sanitize input and return
    return input;
  }
}
```

Thread-starvation issues can be mitigated by choosing a large thread pool size. However, an untrusted caller may still overwhelm the system by supplying more inputs (see guideline "TPS00-J. Use thread pools to enable graceful degradation of service during traffic bursts" on page 121).

Note that operations with further constraints, such as the total number of database connections or total `ResultSet` objects open at a particular time, impose an upper bound on the thread pool size because each thread continues to block until the resource becomes available.

Private static `ThreadLocal` variables may be used to maintain local state in each thread. When using thread pools, the lifetime of `ThreadLocal` variables should be bounded by the corresponding task [Goetz 2006]. Furthermore, these variables should not be used to communicate between tasks. There are additional constraints on the use of `ThreadLocal` variables in thread pools (see guideline "TPS04-J. Ensure ThreadLocal variables are reinitialized when using thread pools" on page 139).

5.2.3 Noncompliant Code Example (Subtasks)

This noncompliant code example contains a series of subtasks that execute in a shared thread pool [Gafter 2006]. The `BrowserManager` class calls `perUser()`, which starts tasks that invoke `perProfile()`. The `perProfile()` method starts tasks that invoke `perTab()`, and, in turn, `perTab()` starts tasks that invoke `doSomething()`. `BrowserManager` then waits for the tasks to finish. The threads are allowed to invoke `doSomething()` in any order, provided `count` correctly records the number of methods executed.

```
public final class BrowserManager {
  private final ExecutorService pool = Executors.newFixedThreadPool(10);
  private final int numberOfTimes;
  private static AtomicInteger count = new AtomicInteger(); // count = 0

  public BrowserManager(int n) {
    numberOfTimes = n;
  }

  public void perUser() {
    methodInvoker(numberOfTimes, "perProfile");
    pool.shutdown();
  }

  public void perProfile() {
    methodInvoker(numberOfTimes, "perTab");
  }

  public void perTab() {
    methodInvoker(numberOfTimes, "doSomething");
  }

  public void doSomething() {
    System.out.println(count.getAndIncrement());
  }
```

```
  public void methodInvoker(int n, final String method) {
    final BrowserManager manager = this;
    Callable<Object> callable = new Callable<Object>() {
      @Override public Object call() throws Exception {
        Method meth = manager.getClass().getMethod(method);
        return meth.invoke(manager);
      }
    };

    Collection<Callable<Object>> collection = Collections.nCopies(n, callable);
    try {
      Collection<Future<Object>> futures = pool.invokeAll(collection);
    } catch (InterruptedException e) {
      // Forward to handler
      Thread.currentThread().interrupt(); // Reset interrupted status
    }
    // ...
  }

  public static void main(String[] args) {
    BrowserManager manager = new BrowserManager(5);
    manager.perUser();
  }
}
```

Unfortunately, this program is susceptible to a thread-starvation deadlock. For example, if each of the five `perUser` tasks spawns five `perProfile` tasks, which each spawn a `perTab` task, the thread pool will be exhausted, and `perTab()` will not be able to allocate any additional threads to invoke the `doSomething()` method.

5.2.4 Compliant Solution (`CallerRunsPolicy`)

This compliant solution selects and schedules tasks for execution, avoiding thread-starvation deadlock. It sets the `CallerRunsPolicy` on a `ThreadPoolExecutor` and uses a `SynchronousQueue` [Gafter 2006]. The policy dictates that if the thread pool runs out of available threads, any subsequent tasks will run in the thread that submitted the tasks.

```
public final class BrowserManager {
  private final static ThreadPoolExecutor pool =
    new ThreadPoolExecutor(0, 10, 60L, TimeUnit.SECONDS,
                           new SynchronousQueue<Runnable>());
  private final int numberOfTimes;
  private static AtomicInteger count = new AtomicInteger(); // count = 0

  static {
    pool.setRejectedExecutionHandler(
    new ThreadPoolExecutor.CallerRunsPolicy());
  }

  // ...
}
```

According to Goetz and colleagues [Goetz 2006]

> *A `SynchronousQueue` is not really a queue at all, but a mechanism for managing handoffs between threads. In order to put an element on the `SynchronousQueue`, another thread must already be waiting to accept the handoff. It no thread is waiting but the current pool size is less than the maximum, `ThreadPoolExecutor` creates a new thread; otherwise the task is rejected according to the saturation policy.*

According to the Java API [Sun 2009b], the `CallerRunsPolicy` class is

> *a handler for rejected tasks that runs the rejected task directly in the calling thread of the `execute` method, unless the executor has been shut down, in which case the task is discarded*

In this compliant solution, tasks that have other tasks waiting to accept the handoff are added to the `SynchronousQueue` when the thread pool is full. For example, tasks corresponding to `perTab()` are added to the `SynchronousQueue` because the tasks corresponding to `perProfile()` are waiting to receive the handoff. Once the pool is full, additional tasks are rejected, according to the saturation policy in effect. Because the `CallerRunsPolicy` is used to handle these rejected tasks, all the rejected tasks are executed in the main thread that started the initial tasks. When all the threads corresponding to `perTab()` have finished executing, the next set of tasks corresponding to `perProfile()` are added to the `SynchronousQueue` because the handoff is subsequently used by the `perUser()` tasks.

The `CallerRunsPolicy` allows the graceful degradation of service when faced with many requests by distributing the workload from the thread pool to the work queue. Because the submitted tasks do not block for any reason other than waiting for other tasks to complete, the policy guarantees that the current thread can handle multiple tasks sequentially. The policy would not

prevent thread-starvation deadlock if the tasks were to block for some other reason, such as network I/O. Furthermore, because `SynchronousQueue` does not store tasks indefinitely for future execution, there is no unbounded queue growth, and all tasks are handled by the current thread or a thread in the thread pool.

This compliant solution is subject to the vagaries of the thread scheduler, which may not schedule the tasks optimally. However, it avoids thread-starvation deadlock.

5.2.5 Risk Assessment

Executing interdependent tasks in a thread pool can lead to denial of service.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
TPS01- J	low	probable	medium	**P4**	**L3**

5.2.6 References

[Gafter 2006] A Thread Pool Puzzler

[Goetz 2006] Section 8.3.2, “Managing queued tasks”
Section 8.3.3, “Saturation Policies”
Section 5.3.3, “Deques and work stealing”

[Sun 2009b]

5.3 TPS02-J. Ensure that tasks submitted to a thread pool are interruptible

Do not submit tasks that do not support interruption using `Thread.interrupt()` to a thread pool if it is necessary to shut down the thread pool or cancel individual tasks within it.

According to the Java API interface [Sun 2009b], the `java.util.concurrent.ExecutorService.shutdownNow()` method

> *Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution. There are no guarantees beyond best-effort attempts to stop processing actively executing tasks. For example, typical implementations will cancel via `Thread.interrupt()`, so any task that fails to respond to interrupts may never terminate.*

Similarly, when attempting to cancel individual tasks within the thread pool using the `Future.cancel()` method, ensure that the tasks support interruption.

5.3.1 Noncompliant Code Example (Shutting Down Thread Pools)

This noncompliant code example submits the `SocketReader` class as a task to the thread pool declared in `PoolService`.

```
public final class SocketReader implements Runnable { // Thread-safe class
  private final Socket socket;
  private final BufferedReader in;
  private final Object lock = new Object();

  public SocketReader(String host, int port) throws IOException {
    this.socket = new Socket(host, port);
    this.in = new BufferedReader(new InputStreamReader(this.socket.getInputStream()));
  }

  // Only one thread can use the socket at a particular time
  @Override public void run() {
    try {
      synchronized (lock) {
        readData();
      }
    } catch (IOException ie) {
      // Forward to handler
    }
  }

  public void readData() throws IOException {
    String string;
    try {
      while ((string = in.readLine()) != null) {
        // Blocks until end of stream (null)
      }
    } finally {
```

```
      shutdown();
    }
  }

  public void shutdown() throws IOException {
    socket.close();
  }
}

public final class PoolService {
  private final ExecutorService pool;

  public PoolService(int poolSize) {
    pool = Executors.newFixedThreadPool(poolSize);
  }

  public void doSomething() throws InterruptedException, IOException {
    pool.submit(new SocketReader("somehost", 8080));
    // ...
    List<Runnable> awaitingTasks = pool.shutdownNow();
  }

  public static void main(String[] args) throws InterruptedException, IOException {
    PoolService service = new PoolService(5);
    service.doSomething();
  }
}
```

Because the task does not support interruption using the `Thread.interrupt()` method, there is no guarantee that the `shutdownNow()` method will shut down the thread pool. Using the latter does not fix the problem either because it waits until all the executing tasks have finished.

Similarly, tasks that use some mechanism other than `Thread.interrupted()` to determine when to shut down will be unresponsive to `shutdown()` or `shutdownNow()`. For instance, tasks that check a volatile flag to determine whether it is safe to shut down will be unresponsive to these methods. The guideline "THI05-J. Do not use `Thread.stop()` to terminate threads" on page 110 provides more information on using a flag to terminate threads.

5.3.2 Compliant Solution (Submit Interruptible Tasks)

This compliant solution defines an interruptible version of the `SocketReader` class, which is instantiated and submitted to the thread pool.

```
public final class SocketReader implements Runnable {
  private final SocketChannel sc;
  private final Object lock = new Object();

  public SocketReader(String host, int port) throws IOException {
    sc = SocketChannel.open(new InetSocketAddress(host, port));
```

```
  }

  @Override public void run() {
    ByteBuffer buf = ByteBuffer.allocate(1024);
    try {
      synchronized (lock) {
        while (!Thread.interrupted()) {
          sc.read(buf);
          // ...
        }
      }
    } catch (IOException ie) {
      // Forward to handler
    }
  }
}

public final class PoolService {
  // ...
}
```

5.3.3 Exceptions

TPS02-EX1: Short-running tasks that execute without blocking are not required to adhere to this guideline.

5.3.4 Risk Assessment

Submitting tasks that are not interruptible may preclude the thread pool from shutting down and cause denial of service.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
TPS02- J	low	probable	medium	P4	L3

5.3.5 References

[Goetz 2006] Chapter 7, “Cancellation and shutdown”

[Sun 2009b] Interface `ExecutorService`

5.4 TPS03-J. Ensure that tasks executing in a thread pool do not fail silently

Long-running tasks should provide a mechanism for notifying the application upon abnormal termination. Failure to do so does not cause any resource leaks because the threads in the pool are still recycled, but it makes failure diagnosis extremely difficult.

The best way to handle exceptions at the application level is to use an exception handler. The handler can perform diagnostic actions, clean up and shut down the JVM, or simply log the details of the failure.

5.4.1 Noncompliant Code Example (Abnormal Task Termination)

This noncompliant code example consists of the `PoolService` class that encapsulates a thread pool and a runnable `Task` class. The `Task.run()` method can throw runtime exceptions such as `NullPointerException`.

```
final class PoolService {
  private final ExecutorService pool = Executors.newFixedThreadPool(10);

  public void doSomething() {
    pool.execute(new Task());
  }
}

final class Task implements Runnable {
  @Override public void run() {
    // ...
    throw new NullPointerException();
    // ...
  }
}
```

The task does not notify the application when it terminates unexpectedly as a result of the runtime exception. Moreover, it does not use any recovery mechanism. Consequently, if `Task` throws a `NullPointerException`, the exception is ignored.

5.4.2 Compliant Solution (`ThreadPoolExecutor` Hooks)

Task-specific recovery or clean-up actions can be performed by overriding the `afterExecute()` hook of the `java.util.concurrent.ThreadPoolExecutor` class. This hook is called when a task concludes successfully by executing all the statements in its `run()` method or halts because of an exception. (`java.lang.Error` might not be captured on specific implementations. See Bug ID 6450211 for more information [Sun 2008b].) When using this approach, substitute the executor service with a custom `ThreadPoolExecutor` that overrides the `afterExecute()` hook as shown below:

```
final class PoolService {
  // The values have been hard-coded for brevity
  ExecutorService pool = new CustomThreadPoolExecutor(10, 10, 10, TimeUnit.SECONDS,
```

```
                        new ArrayBlockingQueue<Runnable>(10));
  // ...
}

class CustomThreadPoolExecutor extends ThreadPoolExecutor {
  // ... Constructor ...

  @Override
  public void afterExecute(Runnable r, Throwable t) {
    super.afterExecute(r, t);
    if (t != null) {
      // Exception occurred, forward to handler
    }
    // ... Perform task-specific clean-up actions
  }

  @Override
  public void terminated() {
    super.terminated();
    // ... Perform final clean-up actions
  }
}
```

The `terminated()` hook is called after all the tasks have finished executing and the `Executor` has terminated cleanly. This hook can be overridden to release resources acquired by the thread pool, much like a `finally` block.

5.4.3 Compliant Solution (Uncaught Exception Handler)

This compliant solution sets an uncaught exception handler on behalf of the thread pool. A `ThreadFactory` argument is passed to the thread pool during construction. The factory is responsible for creating new threads and setting the uncaught exception handler on their behalf. The `Task` class is unchanged from the noncompliant code example.

```
final class PoolService {
  private static final ThreadFactory factory = new
      ExceptionThreadFactory(new MyExceptionHandler());
  private static final ExecutorService pool =
      Executors.newFixedThreadPool(10, factory);

  public void doSomething() {
    pool.execute(new Task()); // Task is a runnable class
  }

  public static class ExceptionThreadFactory implements ThreadFactory  {
    private static final ThreadFactory defaultFactory =
        Executors.defaultThreadFactory();
    private final Thread.UncaughtExceptionHandler handler;
```

```
    public ExceptionThreadFactory(Thread.UncaughtExceptionHandler handler) {
      this.handler = handler;
    }

    @Override public Thread newThread(Runnable run) {
      Thread thread = defaultFactory.newThread(run);
      thread.setUncaughtExceptionHandler(handler);
      return thread;
    }
  }

  public static class MyExceptionHandler extends ExceptionReporter
    implements Thread.UncaughtExceptionHandler {
    // ...

    @Override public void uncaughtException(Thread thread, Throwable t) {
      // Recovery or logging code
    }
  }
}
```

The `ExecutorService.submit()` method can be used to submit a task to a thread pool instead of the `execute()` method to obtain a `Future` object. Note that the uncaught exception handler is not called if `ExecutorService.submit()` is invoked. This is because the thrown exception is considered to be part of the return status and is consequently wrapped in an `ExecutionException` and re-thrown by the `Future.get()` method [Goetz 2006].

5.4.4 Compliant Solution (Future<V> and submit())

This compliant solution invokes the `ExecutorService.submit()` method to submit the task so that a `Future` object can be obtained. It uses the `Future` object to let the task re-throw the exception so that it can be handled locally.

```
final class PoolService {
  private final ExecutorService pool = Executors.newFixedThreadPool(10);

  public void doSomething() {
    Future<?> future = pool.submit(new Task());

    // ...

    try {
      future.get();
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt(); // Reset interrupted status
    } catch (ExecutionException e) {
      Throwable exception = e.getCause();
      // Forward to exception reporter
    }
  }
}
```

Furthermore, any exception that prevents `doSomething()` from obtaining the `Future` value can be handled as required.

5.4.5 Exceptions

TPS03-EX1: This guideline may be violated if the code for all runnable and callable tasks has been audited to ensure that no exceptional conditions are possible. Nonetheless, it is usually a good practice to install a task-specific or global exception handler to initiate recovery or log the exceptional condition.

5.4.6 Risk Assessment

Failing to provide a mechanism for reporting that tasks in a thread pool failed as a result of an exceptional condition can make it harder to find the source of the issue.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
TPS03- J	low	probable	medium	P4	L3

5.4.7 References

[Goetz 2006] Chapter 7.3, “ Handling abnormal thread termination”

[Sun 2009b] Interfaces `ExecutorService`, `ThreadFactory` and class `Thread`

5.5 TPS04-J. Ensure ThreadLocal variables are reinitialized when using thread pools

The `java.lang.ThreadLocal<T>` class provides thread-local variables. According to the Java API [Sun 2009b]

> *These variables differ from their normal counterparts in that each thread that accesses one (via its `get` or `set` method) has its own, independently initialized copy of the variable. ThreadLocal instances are typically `private static` fields in classes that wish to associate state with a thread (e.g., a user ID or Transaction ID).*

The use of `ThreadLocal` objects requires care in classes whose objects are required to be executed by multiple threads in a thread pool. The technique of thread pooling allows threads to be reused when thread creation overhead is too expensive or when creating an unbounded number of threads can diminish the reliability of the system. Every thread that enters the pool expects to see an object in its initial, default state. However, when `ThreadLocal` objects are modified from a thread that is subsequently made available for reuse, the reused thread sees the state of the `ThreadLocal` object as set by the previous thread [Arnold 2006].

5.5.1 Noncompliant Code Example

This noncompliant code example consists of an enumeration of days (`Day`) and two classes (`Diary` and `DiaryPool`). The `Diary` class uses a `ThreadLocal` variable to store thread-specific information, such as each thread's current day. The initial value of the current day is Monday; this can be changed later by invoking the `setDay()` method. The class also contains a `threadSpecificTask()` instance method that performs a thread-specific task.

The `DiaryPool` class consists of the `doSomething1()` and `doSomething2()`methods that each start a thread. The `doSomething1()` method changes the initial (default) value of the day to Friday and invokes `threadSpecificTask()`. On the other hand, `doSomething2()` relies on the initial value of the day (Monday) diary and invokes `threadSpecificTask()`. The `main()` method creates one thread using `doSomething1()` and two more using `doSomething2()`.

```
public enum Day {
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;
}

public final class Diary {
  private static final ThreadLocal<Day> days =
    new ThreadLocal<Day>() {
      // Initialize to Monday
      protected Day initialValue() {
        return Day.MONDAY;
      }
    };

  private static Day currentDay() {
    return days.get();
```

```
  }

  public static void setDay(Day newDay) {
    days.set(newDay);
  }

  // Performs some thread-specific task
  public void threadSpecificTask() {
    // Do task ...
  }
}

public final class DiaryPool {
  final int NoOfThreads = 2; // Maximum number of threads allowed in pool
  final Executor exec;
  final Diary diary;

  DiaryPool() {
    exec = (Executor) Executors.newFixedThreadPool(NoOfThreads);
    diary = new Diary();
  }

  public void doSomething1() {
    exec.execute(new Runnable() {
      @Override public void run() {
        Diary.setDay(Day.FRIDAY);
        diary.threadSpecificTask();
      }
    });
  }

  public void doSomething2() {
    exec.execute(new Runnable() {
      @Override public void run() {
        diary.threadSpecificTask();
      }
    });
  }

  public static void main(String[] args) {
    DiaryPool dp = new DiaryPool();
    dp.doSomething1(); // Thread 1, requires current day as Friday
    dp.doSomething2(); // Thread 2, requires current day as Monday
    dp.doSomething2(); // Thread 3, requires current day as Monday
  }
}
```

The `DiaryPool` class creates a thread pool that reuses a fixed number of threads operating off a shared, unbounded queue. At any point, at most, `NoOfThreads` threads are actively processing tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available. The thread-local state of the thread persists when a thread is recycled.

The following table shows a possible execution order:

Time	Task	Pool Thread	Submitted By Method	Day
1	t_1	1	doSomething1()	Friday
2	t_2	2	doSomething2()	Monday
3	t_3	1	doSomething2()	Friday

In this execution order, the two tasks (t_2 and t_3) that started using `doSomething2()` are expected to observe the current day as Monday. However, because pool thread 1 is reused, t_3 observes the day to be Friday.

5.5.2 Noncompliant Code Example (Increase Thread Pool Size)

This noncompliant code example increases the size of the thread pool from two to three in an attempt to mitigate the issue.

```
public final class DiaryPool {
  final int NoOfThreads = 3;
  // ...
}
```

Although increasing the size of the thread pool resolves the problem for this example, it is not a scalable solution because changing the thread pool size is insufficient when more tasks can be submitted to the pool.

5.5.3 Compliant Solution (`try-finally` Clause)

This compliant solution adds the `removeDay()` method to the `Diary` class and wraps the statements in the `doSomething1()` method of the `DiaryPool` class in a `try-finally` block. The `finally` block restores the initial state of the thread-local `days` object by removing the current thread's value from it.

```
public final class Diary {
  // ...
  public static void removeDay() {
    days.remove();
  }
}

public final class DiaryPool {
  // ...

  public void doSomething1() {
```

```
    exec.execute(new Runnable() {
      @Override public void run() {
          try {
          Diary.setDay(Day.FRIDAY);
          diary.threadSpecificTask();
          } finally {
            Diary.removeDay(); // Diary.setDay(Day.MONDAY) can also be used
          }
      }
    });
  }

  // ...
}
```

If the thread-local variable is read by the same thread again, it is reinitialized using the `initialValue()` method, unless the thread has already set the variable's value explicitly [Sun 2009b]. This solution transfers the responsibility for maintenance to the client (`DiaryPool`) but is a good option when the `Diary` class cannot be modified.

5.5.4 Compliant Solution (`beforeExecute()`)

This compliant solution uses a custom `ThreadPoolExecutor` that extends `ThreadPoolExecutor` and overrides the `beforeExecute()` method. That method is invoked before the `Runnable` task is executed in the specified thread. The method reinitializes the thread-local variable before task `r` is executed by thread `t`.

```
class CustomThreadPoolExecutor extends ThreadPoolExecutor {
  public CustomThreadPoolExecutor(int corePoolSize, int maximumPoolSize,
     long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue) {
        super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue);
  }

  @Override
  public void beforeExecute(Thread t, Runnable r) {
    if (t == null || r == null) {
      throw new NullPointerException();
    }
    Diary.setDay(Day.MONDAY);
    super.beforeExecute(t, r);
  }
}
```

```
public final class DiaryPool {
  // ...
  DiaryPool() {
    exec = new CustomThreadPoolExecutor(NoOfThreads, NoOfThreads,
             10, TimeUnit.SECONDS, new ArrayBlockingQueue<Runnable>(10));
    diary = new Diary();
  }
  // ...
}
```

5.5.5 Exceptions

TPS04-EX1: There is no need to reinitialize a `ThreadLocal` object that does not change state after initialization. For example, there may be only one type of database connection represented by the initial value of the `ThreadLocal` object.

5.5.6 Risk Assessment

Objects using `ThreadLocal` data and executed by different threads in a thread pool without reinitialization might be in an unexpected state when reused.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
TPS04- J	medium	probable	high	P4	L3

5.5.7 References

[Arnold 2006] Section 14.13, “ThreadLocal Variables”

[Sun 2009b] class `java.lang.ThreadLocal<T>`

6 Thread-Safety Miscellaneous (TSM) Guidelines

6.1 TSM00-J. Do not override thread-safe methods with methods that are not thread-safe

Overriding thread-safe methods with methods that are not thread-safe can result in improper synchronization, if the client inadvertently operates on an instance of the subclass. An overridden synchronized method's contract can be violated, if a subclass provides an implementation that is not safe for concurrent use.

Overriding thread-safe methods with methods that are not thread-safe is not, in itself, an error. However, it is disallowed by this guideline because it may easily result in errors that are difficult to diagnose.

The locking strategy of classes designed for inheritance should always be documented. This information can subsequently be used to determine an appropriate locking strategy for subclasses (see guideline "LCK00-J. Use private final lock objects to synchronize classes that may interact with untrusted code" on page 41).

6.1.1 Noncompliant Code Example (Synchronized Method)

This noncompliant code example overrides the synchronized `doSomething()` method in the `Base` class with an unsynchronized method in the `Derived` subclass.

```
class Base {
  public synchronized void doSomething() {
    // ...
  }
}

class Derived extends Base {
  @Override public void doSomething() {
    // ...
  }
}
```

The `doSomething()` method of the `Base` class can be used safely by multiple threads, but instances of the `Derived` subclass cannot.

This programming error can be difficult to diagnose because threads that accept instances of `Base` can also accept instances of its subclasses. Consequently, clients could be unaware that they are operating on an instance of the subclass of a thread-safe class that is not thread-safe.

6.1.2 Compliant Solution (Synchronized Method)

This compliant solution synchronizes the `doSomething()` method of the subclass.

```
class Base {
  public synchronized void doSomething() {
    // ...
  }
}

class Derived extends Base {
  @Override public synchronized void doSomething() {
    // ...
  }
}
```

This compliant solution does not violate guideline "LCK00-J. Use private final lock objects to synchronize classes that may interact with untrusted code" on page 41 because the accessibility of the class is package-private. That type of accessibility is allowable when untrusted code cannot infiltrate the package.

6.1.3 Compliant Solution (Private Final Lock Object)

This compliant solution ensures that the `Derived` class is thread-safe by overriding the synchronized `doSomething()` method of the `Base` class with a method that synchronizes on a private final lock object.

```
class Base {

  public synchronized void doSomething() {
    // ...
  }
}

class Derived extends Base {
  private final Object lock = new Object();

  @Override public void doSomething() {
    synchronized (lock) {
      // ...
    }
  }
}
```

This is an acceptable solution, provided the `Derived` class has a consistent locking policy.

6.1.4 Noncompliant Code Example (Private Lock)

This noncompliant code example defines a `doSomething()` method in the `Base` class that uses a private final lock, in accordance with guideline "LCK00-J. Use private final lock objects to synchronize classes that may interact with untrusted code" on page 41.

```
class Base {
  private final Object lock = new Object();

  public void doSomething() {
    synchronized (lock) {
      // ...
    }
  }
}

class Derived extends Base {
  @Override public void doSomething() {
    try {
      super.doSomething();
    } finally {
      logger.log(Level.FINE, "Did something");
    }
  }
}
```

It is possible for multiple threads to cause the entries to be logged in an order that differs from the order in which the tasks are performed. Consequently, the `doSomething()` method of the `Derived` class cannot be used safely by multiple threads because it is not thread-safe.

6.1.5 Compliant Solution (Private Lock)

This compliant solution synchronizes the `doSomething()` method of the subclass using a private final lock object.

```
class Base {
  private final Object lock = new Object();

  public void doSomething() {
    synchronized (lock) {
      // ...
    }
  }
}

class Derived extends Base {
  private final Object lock = new Object();

  @Override public void doSomething() {
```

```
    synchronized (lock) {
      try {
        super.doSomething();
      } finally {
        logger.log(Level.FINE, "Did something");
      }
    }
  }
}
```

Note that the `Base` and `Derived` objects maintain distinct locks that are inaccessible from each others' classes. Consequently, `Derived` can provide thread-safety guarantees independent of `Base`.

6.1.6 Risk Assessment

Overriding thread-safe methods with methods that are not thread-safe can result in unexpected behavior.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
TSM00- J	low	probable	medium	P4	L3

6.1.7 References

[Sun 2009b]

[Sun 2008b] Sun bug database, Bug ID 4294756

6.2 TSM01-J. Do not let the "this" reference escape during object construction

According to the *Java Language Specification* [Gosling 2005], Section 15.8.3, "`this`"

> *When used as a primary expression, the keyword `this` denotes a value that is a reference to the object for which the instance method was invoked (§15.12), or to the object being constructed. The type of `this` is the class `C` within which the keyword `this` occurs. At run time, the class of the actual object referred to may be the class `C` or any subclass of `C`.*

The `this` reference is said to have escaped when it is made available beyond its current scope. Common ways by which the `this` reference can escape include

- returning `this` from a non-private, overridable method that is invoked from the constructor of a class whose object is being constructed. (For more information, see guideline "MET04-J. Ensure that constructors do not call overridable methods."[10])
- returning `this` from a non-private method of a mutable class, which allows the caller to manipulate the object's state indirectly. This commonly occurs in method-chaining implementations; see guideline "VNA04-J. Ensure that calls to chained methods are atomic" on page 29 for more information.
- passing `this` as an argument to an alien method invoked from the constructor of a class whose object is being constructed
- using inner classes. An inner class implicitly holds a reference to the instance of its outer class, unless the inner class is declared static.
- publishing by assigning `this` to a public static variable from the constructor of a class whose object is being constructed
- overriding the finalizer of a non-final class and obtaining the `this` reference of a partially initialized instance, when the construction of the object ceases. (For more information, see guideline "OBJ04-J. Do not allow partially initialized objects to be accessed."[10]) This can happen when the constructor throws an exception. Misuse is not limited to untrusted code; trusted code can also inadvertently add a finalizer and let `this` escape by violating guideline "OBJ08-J. Avoid using finalizers."[10]
- passing internal object state to an alien method. This enables the method to retrieve the `this` reference of the internal member object.

This guideline describes the potential consequences of allowing the `this` reference to escape during object construction, including race conditions and improper initialization. For example, declaring a field final ensures that all threads see it in a fully initialized state only when the `this` reference does not escape during the corresponding object's construction. Guideline "TSM03-J. Do not publish partially initialized objects" on page 162 describes the guarantees provided by various mechanisms for safe publication and relies on conformance to this guideline. In general, it is important to detect cases where the `this` reference can leak out beyond the scope of the current context. In particular, `public` variables and methods should be carefully scrutinized.

[10] This guideline is described at https://www.securecoding.cert.org/confluence/display/java/.

6.2.1 Noncompliant Code Example (Publish Before Initialization)

This noncompliant code example publishes the `this` reference before initialization has concluded, by storing it in a public static volatile class field.

```
final class Publisher {
  public static volatile Publisher published;
  int num;

  Publisher(int number) {
    published = this;
    // Initialization
    this.num = number;
    // ...
  }
}
```

Consequently, other threads may obtain a partially initialized `Publisher` instance. Also, if the object initialization (and consequently, its construction) depends on a security check within the constructor, the security check can be bypassed if an untrusted caller obtains the partially initialized instance. (For more information, see guideline "OBJ04-J. Do not allow partially initialized objects to be accessed."[11])

6.2.2 Noncompliant Code Example (Non-Volatile Public Static Field)

This noncompliant code example publishes the `this` reference in the last statement of the constructor but is still vulnerable because the `published` field is not declared volatile and has public accessibility.

```
final class Publisher {
  public static Publisher published;
  int num;

  Publisher(int number) {
    // Initialization
    this.num = number;
    // ...
    published = this;
  }
}
```

Because the field is non-volatile and non-final, the statements within the constructor can be reordered by the compiler in such a way that the `this` reference is published before the initialization statements have executed.

[11] This guideline is described at https://www.securecoding.cert.org/confluence/display/java/.

6.2.3 Compliant Solution (Volatile Field and Publish After Initialization)

This compliant solution declares the `published` field volatile and reduces its accessibility to package-private so that callers outside the current package scope cannot obtain the `this` reference.

```
final class Publisher {
  static volatile Publisher published;
  int num;

  Publisher(int number) {
    // Initialization
    this.num = number;
    // ...
    published = this;
  }
}
```

The constructor publishes the `this` reference after initialization has concluded. However, the caller that instantiates `Publisher` must ensure that it does not see the default value of the `num` field before it is initialized (a violation of guideline “TSM03-J. Do not publish partially initialized objects” on page 162). Consequently, the field that holds the reference to `Publisher` might need to be declared volatile in the caller.

Initialization statements may be reordered if the `published` field is not declared volatile. The Java compiler, however, does not allow fields to be declared both volatile and final.

The class `Publisher` must also be final; otherwise, a subclass can call its constructor and publish the `this` reference before the subclass’s initialization has concluded.

6.2.4 Compliant Solution (Public Static Factory Method)

This compliant solution eliminates the internal member field and provides a `newInstance()` factory method that creates and returns a `Publisher` instance.

```
final class Publisher {
  final int num;

  private Publisher(int number) {
    // Initialization
    this.num = number;
  }

  public static Publisher newInstance(int number) {
    Publisher published = new Publisher(number);
    return published;
  }
}
```

This approach ensures that threads do not see an inconsistent `Publisher` instance. The `num` field is also declared final, making the class immutable and eliminating the possibility of obtaining a partially initialized object.

6.2.5 Noncompliant Code Example (Handlers)

This noncompliant code example defines the `ExceptionReporter` interface:

```
public interface ExceptionReporter {
  public void setExceptionReporter(ExceptionReporter er);
  public void report(Throwable exception);
}
```

This interface is implemented by the `DefaultExceptionReporter` class, which reports exceptions after filtering out any sensitive information. (For more information, see guideline "EXC01-J. Use a class dedicated to reporting exceptions."[12])

The `DefaultExceptionReporter` constructor prematurely publishes the `this` reference before construction of the object has concluded. This occurs in the last statement of the constructor (`er.setExceptionReporter(this)`), which sets the exception reporter. Because it is the last statement of the constructor, this may be misconstrued as benign.

```
// Class DefaultExceptionReporter
public class DefaultExceptionReporter implements ExceptionReporter {
  public DefaultExceptionReporter(ExceptionReporter er) {
    // Carry out initialization
    // Incorrectly publishes the "this" reference
    er.setExceptionReporter(this);
  }

  // Implementation of setExceptionReporter() and report()
}
```

The `MyExceptionReporter` class subclasses `DefaultExceptionReporter` with the intent of adding a logging mechanism that logs critical messages before an exception is reported.

```
// Class MyExceptionReporter derives from DefaultExceptionReporter
public class MyExceptionReporter extends DefaultExceptionReporter {
  private final Logger logger;
  public MyExceptionReporter(ExceptionReporter er) {
    super(er); // Calls superclass's constructor
    logger = Logger.getLogger("com.organization.Log"); // Obtain the default logger
  }
  public void report(Throwable t) {
    logger.log(Level.FINEST,"Loggable exception occurred", t);
  }
}
```

[12] This guideline is described at https://www.securecoding.cert.org/confluence/display/java/.

Its constructor invokes the `DefaultExceptionReporter` superclass's constructor (a mandatory first step), which publishes the exception reporter before the initialization of the subclass has concluded. Note that the subclass initialization consists of obtaining an instance of the default logger. Publishing the exception reporter is equivalent to setting it to receive and handle exceptions from that point on.

If an exception occurs before the call to `Logger.getLogger()` in the `MyExceptionReporter` subclass, it is not logged. Instead, a `NullPointerException` is generated, which may, itself, be consumed by the reporting mechanism without being logged.

This erroneous behavior results from the race condition between an oncoming exception and the initialization of `MyExceptionReporter`. If the exception comes too soon, it finds `MyExceptionReporter` in an inconsistent state. This behavior is especially counterintuitive because `logger` is declared final and is not expected to contain an uninitialized value.

This problem can also occur when an event listener is published prematurely. Consequently, it starts receiving event notifications even before the subclass's initialization has concluded.

6.2.6 Compliant Solution

Instead of publishing the `this` reference from the `DefaultExceptionReporter` constructor, this compliant solution adds the `publishExceptionReporter()` method to `DefaultExceptionReporter` to set the exception reporter. This method can be invoked on a subclass instance, after the subclass's initialization has concluded.

```
public class DefaultExceptionReporter implements ExceptionReporter {
  public DefaultExceptionReporter(ExceptionReporter er) {
    // ...
  }

  // Should be called after subclass's initialization is over
  public void publishExceptionReporter() {
    setExceptionReporter(this); // Registers this exception reporter
  }

  // Implementation of setExceptionReporter() and report()
}
```

The `MyExceptionReporter` subclass inherits the `publishExceptionReporter()` method, and a caller who instantiates `MyExceptionReporter` can use its instance to set the exception reporter, after initialization is over.

```
// Class MyExceptionReporter derives from DefaultExceptionReporter
public class MyExceptionReporter extends DefaultExceptionReporter {
  private final Logger logger;

  public MyExceptionReporter(ExceptionReporter er) {
    super(er); // Calls superclass's constructor
    logger = Logger.getLogger("com.organization.Log");
```

```
  }
  // Implementations of publishExceptionReporter(), setExceptionReporter() and
  // report() are inherited
}
```

This approach ensures that the reporter cannot be set before the constructor has fully initialized the subclass and enabled logging.

6.2.7 Noncompliant Code Example (Inner Class)

Inner classes maintain a copy of the `this` reference of the outer object. Consequently, the `this` reference may leak outside the scope [Goetz 2002]. This noncompliant code example uses a different implementation of the `DefaultExceptionReporter` class. The constructor uses an anonymous inner class to publish a `filter()` method.

```
public class DefaultExceptionReporter implements ExceptionReporter {
  public DefaultExceptionReporter(ExceptionReporter er) {
    er.setExceptionReporter(new DefaultExceptionReporter(er) {
      public void report(Throwable t) {
        filter(t);
      }
    });
  }
  // Default implementations of setExceptionReporter() and report()
}
```

The `this` reference of the outer class is published by the inner class so that other threads can see it. Furthermore, if the class is subclassed, the issue described in the noncompliant code example for handlers resurfaces.

6.2.8 Compliant Solution

A `private` constructor alongside a public static factory method can safely publish the `filter()` method from within the constructor [Goetz 2006].

```
public class DefaultExceptionReporter implements ExceptionReporter {
  private final DefaultExceptionReporter defaultER;

  private DefaultExceptionReporter(ExceptionReporter excr) {
    defaultER = new DefaultExceptionReporter(excr) {
      public void report(Throwable t) {
        filter(t);
      }
    };
  }

  public static DefaultExceptionReporter newInstance(ExceptionReporter excr) {
    DefaultExceptionReporter der = new DefaultExceptionReporter(excr);
    excr.setExceptionReporter(der.defaultER);
    return der;
```

```
  }
  // Default implementations of setExceptionReporter() and report()
}
```

Because the constructor is private, untrusted code cannot create instances of the class, prohibiting the `this` reference from escaping. Using a public static factory method to create new instances also protects against publication of partially initialized objects (see guideline “TSM03-J. Do not publish partially initialized objects” on page 162) and untrusted manipulation of internal object state.

6.2.9 Noncompliant Code Example (Thread)

This noncompliant code example starts a thread from within the constructor.

```
final class ThreadStarter implements Runnable {
  public ThreadStarter() {
    Thread thread = new Thread(this);
    thread.start();
  }

  @Override public void run() {
    // ...
  }
}
```

The new thread can access the `this` reference of the current object [Goetz 2002, Goetz 2006]. Notably, the `Thread()` constructor is alien to the `ThreadStarter` class.

6.2.10 Compliant Solution (Thread)

This compliant solution creates and starts the thread in a method instead of the constructor.

```
final class ThreadStarter implements Runnable {
  public ThreadStarter() {
    // ...
  }

  public void startThread() {
    Thread thread = new Thread(this);
    thread.start();
  }

  @Override public void run() {
    // ...
  }
}
```

6.2.11 Exceptions

TSM01-EX1: It is safe to create a thread in the constructor, provided the thread is not started until object construction has completed. This is because a call to `start()` on a thread happens-before any actions in the started thread [Gosling 2005].

In this code example, even though a thread referencing `this` is created in the constructor, it is not started until its `start()` method is called from the `startThread()` method [Goetz 2002, Goetz 2006].

```
final class ThreadStarter implements Runnable {
  Thread thread;

  public ThreadStarter() {
    thread = new Thread(this);
  }

  public void startThread() {
    thread.start();
  }

  @Override public void run() {
    // ...
  }
}
```

TSM01-EX2: The `ObjectPreserver` pattern [Grand 2002] described in guideline "TSM02-J. Do not use background threads during class initialization" on page 157 is also a safe exception to this guideline.

6.2.12 Risk Assessment

Allowing the `this` reference to escape may result in improper initialization and runtime exceptions.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
TSM01-J	medium	probable	high	P4	L3

6.2.13 References

[Goetz 2002]

[Goetz 2006] Section 3.2, "Publication and Escape"

[Gosling 2005] Keyword "this"

[Grand 2002] Chapter 5, "Creational Patterns, Singleton"

6.3 TSM02-J. Do not use background threads during class initialization

Starting and using background threads during class initialization can result in class initialization cycles and deadlock. For example, the main thread responsible for performing class initialization can block waiting for the background thread, which, in turn, will wait for the main thread to finish class initialization. This issue can arise, for example, when a database connection is established in a background thread during class initialization [Bloch 2005b].

6.3.1 Noncompliant Code Example (Background Thread)

In this noncompliant code example, the `static` initializer starts a background thread as part of class initialization. The background thread attempts to initialize a database connection but needs to wait until all members of the `ConnectionFactory` class, including `dbConnection`, have been initialized.

```
public final class ConnectionFactory {
  private static Connection dbConnection;
  // Other fields ...

  static {
    Thread dbInitializerThread = new Thread(new Runnable() {
      @Override public void run() {
        // Initialize the database connection
        try {
          dbConnection = DriverManager.getConnection("connection string");
        } catch (SQLException e) {
          dbConnection = null;
        }
      }
    });

    // Other initialization, for example, start other threads

    dbInitializerThread.start();
    try {
      dbInitializerThread.join();
    } catch (InterruptedException ie) {
      throw new AssertionError(ie);
    }
  }

  public static Connection getConnection() {
    if (dbConnection == null) {
      throw new IllegalStateException("Error initializing connection");
    }
    return dbConnection;
  }

  public static void main(String[] args) {
```

```
    // ...
    Connection connection = getConnection();
  }
}
```

Statically initialized fields are guaranteed to be fully constructed before they are made visible to other threads. (See guideline “TSM03-J. Do not publish partially initialized objects” on page 162 for more information.) Consequently, the background thread must wait for the main (or foreground) thread to finish initialization before it can proceed. However, the `ConnectionFactory` class’s main thread invokes the `join()` method, which waits for the background thread to finish. This interdependency causes a class initialization cycle that results in a deadlock situation [Bloch 2005b].

Similarly, it is inappropriate to start threads from constructors. (See guideline “TSM01-J. Do not let the “this” reference escape during object construction” on page 149 for more information.) Creating timers that perform recurring tasks and starting those timers from within the code responsible for initialization introduces liveness issues.

6.3.2 Compliant Solution (`static` Initializer, No Background Threads)

This compliant solution does not spawn any background threads from the `static` initializer. Instead, all fields are initialized in the main thread.

```
public final class ConnectionFactory {
  private static Connection dbConnection;
  // Other fields ...

  static {
    // Initialize a database connection
    try {
      dbConnection = DriverManager.getConnection("connection string");
    } catch (SQLException e) {
      dbConnection = null;
    }
    // Other initialization (do not start any threads)
  }

  // ...
}
```

6.3.3 Compliant Solution (`ThreadLocal`)

This compliant solution initializes the database connection from a `ThreadLocal` object so that every thread can obtain its own instance of the connection.

```
public final class ConnectionFactory {
  private static final ThreadLocal<Connection> connectionHolder
    = new ThreadLocal<Connection>() {
      @Override public Connection initialValue() {
        try {
          Connection dbConnection = DriverManager.getConnection("connection string");
          return dbConnection;
        } catch (SQLException e) {
          return null;
        }
      }
    };

  // Other fields ...

  static {
    // Other initialization (do not start any threads)
  }

  public static Connection getConnection() {
    Connection connection = connectionHolder.get();
    if (connection == null) {
      throw new IllegalStateException("Error initializing connection");
    }
    return connection;
  }

  public static void main(String[] args) {
    // ...
    Connection connection = getConnection();
  }
}
```

The static initializer can be used to initialize any other shared class fields. Alternatively, the fields can be initialized from the `initialValue()` method.

6.3.4 Exceptions

TSM02-EX1: It is permissible to start a background thread during class initialization provided the thread does not access any fields. For example, the `ObjectPreserver` class (based on *Patterns in Java* [Grand 2002]) shown below provides a mechanism for storing object references, which prevents an object from being garbage-collected, even if the object is not de-referenced in the future.

```
public final class ObjectPreserver implements Runnable {
  private static final ObjectPreserver lifeLine = new ObjectPreserver();

  private ObjectPreserver() {
    Thread thread = new Thread(this);
    thread.setDaemon(true);
    thread.start(); // Keep this object alive
  }

  // Neither this class nor HashMap will be garbage-collected.
  // References from HashMap to other objects will also exhibit this property
  private static final ConcurrentHashMap<Integer, Object> protectedMap
    = new ConcurrentHashMap<Integer, Object>();

  public synchronized void run() {
    try {
      wait();
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt(); // Reset interrupted status
    }
  }

  // Objects passed to this method will be preserved until
  // the unpreserveObject() method is called
  public static void preserveObject(Object obj) {
    protectedMap.put(0, obj);
  }

  // Returns the same instance every time
  public static Object getObject() {
    return protectedMap.get(0);
  }

  // Unprotect the objects so that they can be garbage-collected
  public static void unpreserveObject() {
    protectedMap.remove(0);
  }
}
```

This is a singleton class. (See guideline "MSC16-J. Address the shortcomings of the Singleton design pattern"[13] for more information on how to defensively code singleton classes.) The initialization involves creating a background thread using the current instance of the class. The thread waits indefinitely by invoking `Object.wait()`. Consequently, this object persists for the remainder of the JVM's lifetime. Because the object is managed by a daemon thread, the thread does not hinder a normal shutdown of the JVM.

While the initialization does involve a background thread, that thread does not access any fields or create any liveness or safety issues. Consequently, this code is a safe and useful exception to this guideline.

6.3.5 Risk Assessment

Starting and using background threads during class initialization can result in deadlock conditions.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
TSM02- J	low	probable	high	P2	L3

6.3.6 References

[Bloch 2005b]

[Grand 2002] Chapter 5, "Creational Patterns, Singleton"

[13] This guideline is described at https://www.securecoding.cert.org/confluence/display/java/.

6.4 TSM03-J. Do not publish partially initialized objects

During initialization of a shared object, the object must only be accessible to the thread constructing it. However, the object can be published safely (that is, made visible to other threads) once it is initialized. The JMM allows multiple threads to observe the object after its initialization has begun, but before it has concluded. Consequently, it is important to ensure that a partially initialized object is not published.

This guideline prohibits publishing a reference to a partially initialized member object instance before initialization has concluded. Guideline "TSM01-J. Do not let the "this" reference escape during object construction" on page 149 prohibits the `this` reference of the current object from escaping.

6.4.1 Noncompliant Code Example

This noncompliant code example constructs a `Helper` object in the `initialize()` method of the `Foo` class. The `Helper` object's fields are initialized by its constructor.

```
class Foo {
  private Helper helper;

  public Helper getHelper() {
    return helper;
  }

  public void initialize() {
    helper = new Helper(42);
  }
}

public class Helper {
  private int n;

  public Helper(int n) {
    this.n = n;
  }
  // ...
}
```

If a thread accesses `helper` using the `getHelper()` method before the `initialize()` method has been called, the thread will observe an uninitialized `helper` field. Later, if one thread calls `initialize()` and another calls `getHelper()`, the second thread might observe one of the following:

- the `helper` reference as NULL
- a fully initialized `Helper` object with the `n` field set to 42
- a partially initialized `Helper` object with an uninitialized `n` that contains the default value `0`

In particular, the JMM permits compilers to allocate memory for the new `Helper` object and assign it to the `helper` field before initializing it. In other words, the compiler can reorder the write to the `helper` instance field with the write that initializes the `Helper` object (that is, `this.n = n`) such that the former occurs first. This exposes a race window during which other threads may observe a partially initialized `Helper` object instance.

There is a separate issue: if two threads call `initialize()`, two `Helper` objects are created. This is a performance issue and not a correctness issue because `n` will be properly initialized and the unused `Helper` objects will be garbage-collected.

6.4.2 Compliant Solution (Synchronization)

The publication of partially constructed object references can be prevented by using method synchronization, as shown in this compliant solution.

```
class Foo {
  private Helper helper;

  public synchronized Helper getHelper() {
    return helper;
  }

  public synchronized void initialize() {
    helper = new Helper(42);
  }
}
```

Synchronizing both methods guarantees that they will not execute concurrently. If one thread calls `initialize()` just before another thread calls `getHelper()`, the synchronized `initialize()` method will always finish first. The `synchronized` keyword establishes a happens-before relationship between the two threads. This guarantees that the thread calling `getHelper()` sees the fully initialized `Helper` object or none at all (that is, `helper` contains a `null` reference). This approach guarantees proper publication for both immutable and mutable members.

6.4.3 Compliant Solution (Final Field)

If the `helper` field is declared final, it is guaranteed to be fully constructed before its reference is made visible.

```
class Foo {
  private final Helper helper;

  public Helper getHelper() {
    return helper;
  }

  public Foo() {
    helper = new Helper(42);
  }
}
```

However, this solution requires the assignment of a new `Helper` instance to `helper` from `Foo`'s constructor. According to the *Java Language Specification*, Section 17.5.2, "Reading Final Fields During Construction" [Gosling 2005]

> *A read of a `final` field of an object within the thread that constructs that object is ordered with respect to the initialization of that field within the constructor by the usual happens-before rules. If the read occurs after the field is set in the constructor, it sees the value the `final` field is assigned, otherwise it sees the default value.*

Consequently, the reference to the `Helper` instance should not be published before the `Foo` class's constructor has finished its initialization (see guideline "TSM01-J. Do not let the "this" reference escape during object construction" on page 149).

6.4.4 Compliant Solution (Final Field and Thread-Safe Composition)

Some collection classes provide thread-safe access to contained elements. If the `Helper` object is inserted into such a collection, it is guaranteed to be fully initialized before its reference is made visible. This compliant solution encapsulates the `helper` field in a `Vector<Helper>`.

```
class Foo {
  private final Vector<Helper> helper;

  public Foo() {
    helper = new Vector<Helper>();
  }

  public Helper getHelper() {
    if (helper.isEmpty()) {
      initialize();
    }
    return helper.elementAt(0);
  }
```

```
  public synchronized void initialize() {
    if (helper.isEmpty()) {
      helper.add(new Helper(42));
    }
  }
}
```

The `helper` field is declared final to guarantee that the vector is created before any accesses take place. It can be initialized safely by invoking the synchronized `initialize()` method, which ensures that only one `Helper` object is ever added to the vector. If `getHelper()` is invoked before `initialize()`, it calls `initialize()` to avoid the possibility of a null-pointer dereference by the client. The `getHelper()` method does not require synchronization to simply return `Helper`, and—because the synchronized `initialize()` method also checks to make sure `helper` is empty before adding a new `Helper` object—there is no possibility of exploiting a race condition to add a second object to the vector.

6.4.5 Compliant Solution (Static Initialization)

In this compliant solution, the `helper` field is statically initialized, ensuring that the object referenced by the field is fully initialized before its reference is visible.

```
// Immutable Foo
final class Foo {
  private static final Helper helper = new Helper(42);

  public static Helper getHelper() {
    return helper;
  }
}
```

Although not a requirement, the `helper` field should be declared final to document the class's immutability.

According to *the Java Memory Model and Thread Specification*, Section 9.2.3, "Static Final Fields" [JSR-133 2004]

> *The rules for class initialization ensure that any thread that reads a `static` field will be synchronized with the static initialization of that class, which is the only place where `static final` fields can be set. Thus, no special rules in the JMM are needed for `static final` fields.*

6.4.6 Compliant Solution (Immutable Object - Final Fields, Volatile Reference)

The JMM guarantees that any final fields of an object are fully initialized before a published object becomes visible [Goetz 2006]. By declaring `n` final, the `Helper` class is made immutable. Furthermore, if the `helper` field is declared volatile in compliance with guideline "VNA01-J. Ensure visibility of shared references to immutable objects" on page 13, `Helper`'s reference is guaranteed to be made visible to any thread that calls `getHelper()` after `Helper` has been fully initialized.

```
class Foo {
  private volatile Helper helper;

  public Helper getHelper() {
    return helper;
  }

  public void initialize() {
    helper = new Helper(42);
  }
}

// Immutable Helper
public final class Helper {
  private final int n;

  public Helper(int n) {
    this.n = n;
  }
  // ...
}
```

This compliant solution requires that `helper` be declared volatile and class `Helper` be immutable. If it were not immutable, the code would violate guideline "VNA06-J. Do not assume that declaring an object reference volatile guarantees visibility of its members" on page 35, and additional synchronization would be necessary (see the next compliant solution). And if the `helper` field were non-volatile, it would violate guideline "VNA01-J. Ensure visibility of shared references to immutable objects" on page 13.

Similarly, a public static factory method that returns a new instance of `Helper` can be provided in the `Helper` class. This approach allows the `Helper` instance to be created in a `private` constructor.

6.4.7 Compliant Solution (Mutable Thread-Safe Object, Volatile Reference)

If `Helper` is mutable but thread-safe, it can be published safely by declaring the `helper` field in the `Foo` class volatile.

```
class Foo {
  private volatile Helper helper;
```

```
  public Helper getHelper() {
    return helper;
  }

  public void initialize() {
    helper = new Helper(42);
  }
}

// Mutable but thread-safe Helper
public class Helper {
  private volatile int n;
  private final Object lock = new Object();

  public Helper(int n) {
    this.n = n;
  }

  public void setN(int value) {
    synchronized (lock) {
      n = value;
    }
  }
}
```

Because the `Helper` object can change state after its construction, synchronization is necessary to ensure the visibility of mutable members after initial publication. Consequently, the `setN()` method is synchronized to provide the visibility of the `n` field in this compliant solution (see guideline "VNA06-J. Do not assume that declaring an object reference volatile guarantees visibility of its members" on page 35).

If the `Helper` class is not synchronized properly, declaring `helper` volatile in the `Foo` class only guarantees the visibility of the initial publication of `Helper` and not of subsequent state changes. Consequently, volatile references alone are inadequate for publishing objects that are not thread-safe.

If the `helper` field in the `Foo` class is not declared volatile, the `n` field should be declared volatile so that a happens-before relationship is established between the initialization of `n` and the write of `Helper` to the `helper` field. This is in compliance with guideline "VNA06-J. Do not assume that declaring an object reference volatile guarantees visibility of its members" on page 35. This is required only when the caller (class `Foo`) cannot be trusted to declare `helper` volatile.

Because the `Helper` class is declared public, it uses a private lock to handle synchronization in conformance with guideline "LCK00-J. Use private final lock objects to synchronize classes that may interact with untrusted code" on page 41.

6.4.8 Exceptions

TSM03-EX1: Classes that prevent partially initialized objects from being used may publish partially initialized objects. This may be implemented, for example, by setting a volatile boolean flag in the last statement of the initializing code and ensuring that this flag is set before allowing class methods to execute.

The following compliant solution illustrates this technique:

```
public class Helper {
  private int n;
  private volatile boolean initialized; // Defaults to false

  public Helper(int n) {
    this.n = n;
    this.initialized = true;
  }

  public void doSomething() {
    if (!initialized) {
      throw new SecurityException("Cannot use partially initialized instance");
    }
    // ...
  }
  // ...
}
```

This technique ensures that even if the reference to the `Helper` object instance is published before its initialization is over, the instance is unusable. The instance is unusable because every method within `Helper` must check the flag to determine whether the initialization has finished.

6.4.9 Risk Assessment

Failing to synchronize access to shared mutable data can cause different threads to observe different states of the object or a partially initialized object.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
TSM03-J	medium	probable	medium	P8	L2

6.4.10 References

[Arnold 2006] Section 14.10.2, “Final Fields and Security”

[Bloch 2001] Item 48: “Synchronize access to shared mutable data”

[Goetz 2006] Section 3.5.3, “Safe Publication Idioms”

[Goetz 2006c] Pattern #2: “one-time safe publication”

[Pugh 2004]

[Sun 2009b]

6.5 TSM04-J. Document thread-safety and use annotations where applicable

The Java language annotation facility is useful for documenting design intent. Source code annotation is a mechanism for associating metadata with a program element and making it available to the compiler, analyzers, debuggers, or the JVM for examination. Several annotations are available for documenting thread-safety or the lack thereof.

6.5.1 Obtaining Concurrency Annotations

Two sets of concurrency annotations are freely available and licensed for use in any code. The first set consists of four annotations described in *Java Concurrency in Practice* (JCIP) [Goetz 2006], which can be downloaded at jcip.net (jar, javadoc, source). The JCIP annotations are released under the Creative Commons Attribution License.

The second, larger set of concurrency annotations is available from and supported by SureLogic. These annotations are released under The Apache Software License, Version 2.0 and can be downloaded at surelogic.com (jar, javadoc, source). They can be verified by the SureLogic JSure tool and are useful for documenting code, even if the tool is unavailable. These annotations include the JCIP annotations because they are supported by the JSure tool. (JSure also supports the use of the JCIP JAR file.)

To use the annotations, download and add one or both of the aforementioned JAR files to the code's build path. The use of these annotations to document thread-safety is described in the following sections.

6.5.2 Documenting Intended Thread-Safety

JCIP provides three class-level annotations to describe the programmer's design intent with respect to thread-safety.

The `@ThreadSafe` annotation is applied to a class to indicate that it is thread-safe. This means that no sequences of accesses (reads and writes to public fields, calls to public methods) can leave the object in an inconsistent state, regardless of the interleaving of these accesses by the runtime or any external synchronization or coordination on the part of the caller.

For example, the `Aircraft` class shown below specifies that it is thread-safe as part of its locking policy documentation. This class protects the `x` and `y` fields using a reentrant lock.

```
@ThreadSafe
@Region("private AircraftState")
@RegionLock("StateLock is stateLock protects AircraftState")
public final class Aircraft {
  private final Lock stateLock = new ReentrantLock();
  // ...
  @InRegion("AircraftState")
  private long x, y;
  // ...
  public void setPosition(long x, long y) {
    stateLock.lock();
```

```
    try {
      this.x = x;
      this.y = y;
    } finally {
      stateLock.unlock();
    }
  }
  // ...
}
```

The `@Region` and `@RegionLock` annotations document the locking policy that the promise of thread-safety is predicated on.

Even if one or more `@RegionLock` or `@GuardedBy` annotations have been used to document the locking policy of a class, the `@ThreadSafe` annotation provides an intuitive way for reviewers to learn that the class is thread-safe.

The `@Immutable` annotation is applied to immutable classes. Immutable objects are inherently thread-safe; after they are fully constructed, they may be published via a volatile reference and shared safely among multiple threads.

The following example shows an immutable `Point` class:

```
@Immutable
public final class Point {
  private final int f_x;
  private final int f_y;

  public Point(int x, int y) {
    f_x = x;
    f_y = y;
  }

  public int getX() {
    return f_x;
  }

  public int getY() {
    return f_y;
  }
}
```

According to Bloch [Bloch 2008]

> *It is not necessary to document the immutability of `enum` types. Unless it is obvious from the return type, static factories must document the thread safety of the returned object, as demonstrated by `Collections.synchronizedMap`.*

The `@NotThreadSafe` annotation is applied to classes that are not thread-safe. Many classes fail to document whether they are safe for multithreaded use. Consequently, a programmer has no

easy way to determine whether the class is thread-safe. This annotation provides a clear indication of the class's lack of thread-safety.

For example, most of the collection implementations provided in `java.util` are not thread-safe. The `java.util.ArrayList` class could document this as follows:

```
package java.util.ArrayList;

@NotThreadSafe
public class ArrayList<E> extends ... {
  // ...
}
```

6.5.3 Documenting Locking Policies

It is important to document all the locks that are being used to protect shared state. According to Goetz and colleagues [Goetz 2006]

> *For each mutable state variable that may be accessed by more than one thread,* all *accesses to that variable must be performed with the* same *lock held. In this case, we say that the variable is* guarded by *that lock.*

JCIP provides the `@GuardedBy` annotation for this purpose, while SureLogic provides the `@RegionLock` annotation. The field or method to which the `@GuardedBy` annotation is applied can only be accessed when holding a particular lock. This may be an intrinsic lock or a dynamic lock such as `java.util.concurrent.Lock`.

For example, the following `MovablePoint` class implements a movable point that has the capability of remembering its past locations using the `memo` array list.

```
@ThreadSafe
public final class MovablePoint {

  @GuardedBy("this")
    double xPos = 1.0;
  @GuardedBy("this")
    double yPos = 1.0;
  @GuardedBy("itself")
    static final List<MovablePoint> memo = new ArrayList<MovablePoint>();

  public void move(double slope, double distance) {
    synchronized (this) {
      rememberPoint(this);
      xPos += (1 / slope) * distance;
      yPos += slope * distance;
    }
  }

  public static void rememberPoint(MovablePoint value) {
    synchronized (memo) {
```

```
      memo.add(value);
    }
  }
}
```

The `@GuardedBy` annotations on the `xPos` and `yPos` fields indicate that access to these fields is protected by holding a lock on `this` (as is done in the `move()` method, which modifies these fields). The `@GuardedBy` annotation on the `memo` list indicates that a lock on the `ArrayList` object protects its contents (as is done in the `rememberPoint()` method).

One issue with the `@GuardedBy` annotation is that it does not clarify that there is a relationship between the fields of a class. This limitation can be overcome by using the SureLogic `@RegionLock` annotation, which declares a new region lock for the class to which this annotation is applied. This declaration creates a new named lock that associates a particular lock object with a region of the class. The region may be accessed only when the lock is held.

For example, the `SimpleLock` locking policy indicates that synchronizing on the instance protects all of its state:

```
@RegionLock("SimpleLock is this protects Instance")
class Simple { ... }
```

Unlike `@GuardedBy`, the `@RegionLock` annotation allows the programmer to give an explicit, and hopefully meaningful, name to the locking policy.

In addition to naming the locking policy, the `@Region` annotation allows a name to be given to the region of the state that is being protected. That name makes it clear that the state and locking policy belong together, as demonstrated in the following example:

```
@Region("private AircraftPosition")
@RegionLock("StateLock is stateLock protects AircraftPosition")
public final class Aircraft {
  private final Lock stateLock = new ReentrantLock();

  @InRegion("AircraftPosition")
  private long x, y;

  @InRegion("AircraftPosition")
  private long altitude;
  // ...
  public void setPosition(long x, long y) {
    stateLock.lock();
    try {
      this.x = x;
      this.y = y;
    } finally {
      stateLock.unlock();
    }
  }
```

```
  // ...
}
```

In this example, a locking policy named `StateLock` is used to indicate that locking on `stateLock` protects the named `AircraftPosition` region, which includes the mutable state used to represent the position of the aircraft.

6.5.4 Construction of Mutable Objects

Typically, object construction is considered an exception to the locking policy because objects are thread-confined when they are created. An object is confined to the thread that uses the `new` operator to create its instance. After creation, the object can be published to other threads safely. However, the object is not shared until the thread that created the instance allows it to be shared. Safe publication approaches discussed in guideline "TSM01-J. Do not let the "this" reference escape during object construction" on page 149 can be expressed succinctly with the `@Unique("return")` annotation.

For example, in the code shown below, the `@Unique("return")` annotation documents that the object returned from the constructor is a unique reference.

```
@RegionLock("Lock is this protects Instance")
public final class Example {
  private int x = 1;
  private int y;

  @Unique("return")
  public Example(int y) {
    this.y = y;
  }
  // ...
}
```

6.5.5 Documenting Thread-Confinement Policies

Sutherland and Scherlis propose annotations that can document thread-confinement policies. Their approach allows verification of the annotations against code as it exists [Sutherland 2010].

For example, the following annotations express the design intent that a program has, at most, one AWT event dispatch thread and several Compute threads, and that the Compute threads are forbidden to handle AWT data structures or events:

```
@ColorDeclare AWT, Compute
@IncompatibleColors AWT, Compute
@MaxColorCount AWT 1
```

6.5.6 Documenting Wait-Notify Protocols

According to Goetz and colleagues [Goetz 2006]

> *A state-dependent class should either fully expose (and document) its waiting and notification protocols to subclasses, or prevent subclasses from participating in them at all. (This is an extension of "design and document for inheritance, or else prohibit it" [EJ Item 15].) At the very least, designing a state-dependent class for inheritance requires exposing the condition queues and locks and documenting the condition predicates and synchronization policy; it may also require exposing the underlying state variables. (The worst thing a state-dependent class can do is expose its state to subclasses but not document its protocols for waiting and notification; this is like a class exposing its state variables but not documenting its invariants.).*

Wait-notify protocols should be documented adequately. Currently, we are not aware of any annotations for this purpose.

6.5.7 Risk Assessment

Annotations of concurrent code document the design intent and can be used to automate the detection and prevention of race conditions and data races.

Guideline	Severity	Likelihood	Remediation Cost	Priority	Level
TSM04- J	low	probable	medium	P4	L3

6.5.8 References

[Bloch 2008] Item 70: "Document thread safety"

[Goetz 2006]

[Sutherland 2010]

Appendix Definitions

alien method
"From the perspective of a class C, an alien method is one whose behavior is not fully specified by C. This includes methods in other classes as well as overrideable methods (neither private nor final) in C itself" [Goetz 2006].

atomicity
When applied to an operation on primitive data, indicates that other threads that might access the data might see the data as it exists before the operation occurs or after the operation has completed, but may never see an intermediate value of the data.

canonicalization
Reducing the input to its equivalent simplest known form.

class variable
A class variable is a `static` field associated with the containing class.

condition predicate
A condition predicate is an expression constructed from the state variables of a class that must be true for a thread to continue execution. The thread pauses execution, via `Object.wait()`, `Thread.sleep()`, or some other mechanism, and is resumed later, presumably when the requirement is true and when it is notified [Goetz 2006].

conflicting accesses
Two accesses to (reads of or writes to) the same variable provided that at least one of the accesses is a write. [Gosling 2005].

data race
"Conflicting accesses of the same variable that are not ordered by a happens-before relationship" [Gosling 2005].

deadlock
Two or more threads are said to have deadlocked when both block waiting for each others' locks. Neither thread can make any progress.

happens-before order

"Two actions can be ordered by a happens-before relationship. If one action happens-before another, then the first is visible to and ordered before the second. [. . .] It should be noted that the presence of a happens-before relationship between two actions does not necessarily imply that they have to take place in that order in an implementation. If the reordering produces results consistent with a legal execution, it is not illegal. [. . .] More specifically, if two actions share a happens-before relationship, they do not necessarily have to appear to have happened in that order to any code with which they do not share a happens-before relationship. Writes in one thread that are in a data race with reads in another thread may, for example, appear to occur out of order to those reads" [Gosling 2005].

heap memory

"Memory that can be shared between threads is called shared memory or heap memory. All instance fields, static fields and array elements are stored in heap memory.[...] Local variables (§14.4), formal method parameters (§8.4.1) or exception handler parameters are never shared between threads and are unaffected by the memory model" [Gosling 2005].

immutable

When applied to an object, this means that its state cannot be changed after being initialized.
"An object is immutable if:

- Its state cannot be modified after construction;
- All its fields are final;[12] and
- It is properly constructed (the `this` reference does not escape during construction).

[12] It is technically possible to have an immutable object without all fields being `final`. `String` is such a class but this relies on delicate reasoning about benign data races that requires a deep understanding of the Java Memory Model. (For the curious: `String` lazily computes the hash code the first time `hashCode` is called and caches it in a nonfinal field, but this works only because that field can take on only one nondefault value that is the same every time it is computed because it is derived deterministically from immutable state" [Goetz 2006]. Immutable objects are inherently thread-safe; they may be shared between multiple threads or published without synchronization, though it is usually required to declare the fields containing their references `volatile` to ensure visibility. An immutable object may contain mutable sub-objects, provided the state of the sub-objects cannot be modified after construction of the immutable object has concluded.

initialization safety

"An object is considered to be completely initialized when its constructor finishes. A thread that can only see a reference to an object after that object has been completely initialized is guaranteed to see the correctly initialized values for that object's final fields" [Gosling 2005].

interruption policy

"An interruption policy determines how a thread interprets an interruption request - what it does (if anything) when one is detected, what units of work are considered atomic with respect to interruption, and how quickly it reacts to interruption" [Goetz 2006].

instance variable

An instance variable is a non-static field that is a part of every instance of the class

liveness
Every operation or method invocation executes to completion without interruptions, even if it goes against safety.

memory model
"The rules that determine how memory accesses are ordered and when they are guaranteed to be visible are known as the memory model of the Java programming language" [Arnold 2006]. "A memory model describes, given a program and an execution trace of that program, whether the execution trace is a legal execution of the program" [Gosling 2005].

normalization
Lossy conversion of the data to its simplest known (and anticipated) form. "When implementations keep strings in a normalized form, they can be assured that equivalent strings have a unique binary representation" [Davis 2009].

normalization (URI)
Normalization is the process of removing unnecessary "." and ".." segments from the path component of a hierarchical URI. Each "." segment is simply removed. A ".." segment is removed only if it is preceded by a non-".." segment. Normalization has no effect upon opaque URIs [Sun 2009b].

obsolete reference
"An obsolete reference is simply a reference that will never be dereferenced again" [Bloch 08].

open call
"An alien method invoked outside of a synchronized region is known as an open call" [Lea 2000a Section 2.4.1.3, Bloch 2008].

partial order
An order defined for some, but not necessarily all, pairs of items. For instance, the sets {a, b} and {a, c, d} are subsets of {a, b, c, d}, but neither is a subset of the other. So "subset of" is a partial order on sets. [Black 2004a]

program order
The order that inter-thread actions are performed by a thread according to the intra-thread semantics of the thread. "Program order [can be described] as the order of bytecodes present in the .class file, as they would execute based on control flow values" (David Holmes, JMM Mailing List).

publishing objects
"Publishing an object means making it available to code outside of its current scope, such as by storing a reference to it where other code can find it, returning it from a nonprivate method, or passing it to a method in another class" [Goetz 2006].

race condition
"General races cause nondeterministic execution and are failures in programs intended to be deterministic" [Netzer 1992]. "A race condition occurs when the correctness of a computation depends on the relative timing or interleaving of multiple threads by the runtime" [Goetz 2006].

relativization (URI)

"[Relativization] is the inverse of resolution. For example, relativizing the URI `http://java.sun.com/j2se/1.3/docs/guide/index.html` against the base URI `http://java.sun.com/j2se/1.3` yields the relative URI `docs/guide/index.html`" [Sun 2009b].

safety

Its main goal is to ensure that all objects maintain consistent states in a multithreaded environment [Lea 2000a].

sanitization

Sanitization is a term used for validating input and transforming it to a representation that conforms to the input requirements of a complex subsystem. For example, a database may require all invalid characters to be escaped or eliminated prior to their storage. Input sanitization refers to the elimination of unwanted characters from the input by means of removal, replacement, encoding or escaping the characters.

sequential consistency

"Sequential consistency is a very strong guarantee that is made about visibility and ordering in an execution of a program. Within a sequentially consistent execution, there is a total order over all individual actions (such as reads and writes) which is consistent with the order of the program, and each individual action is atomic and is immediately visible to every thread. [. . .] If a program is correctly synchronized, then all executions of the program will appear to be sequentially consistent (§17.4.3)" [Gosling 2005]. Sequential consistency implies there will be no compiler optimizations in the statements of the action. Adopting sequential consistency as the memory model and disallowing other primitives can be overly restrictive because under this condition, the compiler is not allowed to make optimizations and reorder code [Gosling 2005].

synchronization

"The Java programming language provides multiple mechanisms for communicating between threads. The most basic of these methods is *synchronization*, which is implemented using monitors. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor. Any other threads attempting to lock that monitor are blocked until they can obtain a lock on that monitor" [Gosling 2005].

starvation

A condition wherein one or more threads prevent other threads from accessing a shared resource over extended periods of time. For instance, a thread that invokes a synchronized method, which performs some time-consuming operation, starves other threads.

thread-safe

An object is thread-safe if it can be shared by multiple threads without the possibility of any data races. "A thread-safe object performs synchronization internally, so multiple threads can freely access it through its public interface without further synchronization" [Goetz 2006]. Immutable classes are thread-safe by definition. Mutable classes may also be thread-safe if they are properly synchronized.

total order

An order defined for all pairs of items of a set. For instance, <= (less than or equal to) is a total order on integers, that is, for any two integers, one of them is less than or equal to the other [Black 2004b].

trusted code

Code that is loaded by the primordial class loader, irrespective of whether it constitutes the Java API or not. In this text, this meaning is extended to include code that is obtained from a known entity and given permissions that untrusted code lacks. By this definition, untrusted and trusted code can coexist in the namespace of a single class loader (not necessarily the primordial class loader). In such cases, the security policy must make this distinction clear by assigning appropriate privileges to trusted code, while denying the same from untrusted code.

untrusted code

Code of unknown origin that can potentially cause some harm when executed. Untrusted code may not always be malicious but this is usually hard to determine automatically. Consequently, untrusted code should be run in a sandboxed environment.

volatile

"A write to a volatile field (§8.3.1.4) happens-before every subsequent read of that field" [Gosling 2005]. "Operations on the master copies of volatile variables on behalf of a thread are performed by the main memory in exactly the order that the thread requested" [Sun 1999b]. Accesses to a `volatile` variable are sequentially consistent, which also means that the operations are exempt from compiler optimizations. Declaring a variable `volatile` ensures that all threads see the most up-to-date value of the variable, if any thread modifies it. Volatile guarantees atomic reads and writes of primitive values, however, it does not guarantee the atomicity of composite operations such as variable incrementation (read-modify-write sequence).

vulnerability

"A set of conditions that allows an attacker to violate an explicit or implicit security policy" [Seacord 2005].

Bibliography

URLs are valid as of the publication date of this document.

[Abadi 1996]
Abadi, Martin & Needham, Roger. "Prudent Engineering Practice for Cryptographic Protocols." *IEEE* Transactions *on Software Engineering 22*, 1 (January1996): 6 - 15.

[Apache 2008]
Apache. *Class FunctionTable, Field detail, public static FuncLoader m_functions*. http://www.stylusstudio.com/api/xalan-j_2_6_0/org/apache/xpath/compiler/FunctionTable.htm (2008).

[Apache 2009a]
Apache Software Foundation. *Apache Tomcat 6.0 – Changelog.* http://tomcat.apache.org/tomcat-6.0-doc/changelog.html (2009).

[Apache 2009b]
Apache Software Foundation. *Apache Tomcat 6.x Vulnerabilities.* http://tomcat.apache.org/security-6.html (2009).

[Arnold 2006]
Arnold, Ken, Gosling, James, & Holmes, David. *The Java Programming Language, Fourth Edition*. Addison Wesley Professional, 2006.

[Austin 2000]
Austin, Calvin & Pawlan, Monica. *Advanced Programming for the Java 2 Platform.* Addison Wesley Longman, 2000.

[Black 2004a]
Black, Paul E. & Tananbaum, Paul J. *Partial Order.* http://www.itl.nist.gov/div897/sqg/dads/HTML/partialorder.html (2004).

[Black 2004b]
Black, Paul E. & Tananbaum, Paul J. *Total Order.* http://www.itl.nist.gov/div897/sqg/dads/HTML/totalorder.html (2004).

[Bloch 2001]
Bloch, Joshua. *Effective Java, Programming Language Guide*. Addison Wesley, 2001.

[Bloch 2005a]
Bloch, Joshua & Gafter, Neal. *Java Puzzlers: Traps, Pitfalls, and Corner Cases*. Pearson Education, Inc., 2005.

[Bloch 2005b]
Bloch, Joshua & Gafter, Neal. "Yet More Programming Puzzlers," *Proceedings of the JavaOne Conference.* San Francisco, CA, June 2005.

[Bloch 2007a]
Bloch, Joshua. "Effective Java Reloaded: This Time It's (not) for Real," *Proceedings of the JavaOne Conference.* San Francisco, CA, May 2007.

[Bloch 2008]
Bloch, Joshua. *Effective Java, 2nd edition*. Addison Wesley, 2008.

[Bloch 2009]
Bloch, Joshua & Gafter, Neal. "Return of the Puzzlers: Schlock and Awe," *Proceedings of the JavaOne Conference.* San Francisco, CA, June 2009.

[Boehm 2005]
Boem, Hans J. "Finalization, Threads, and the Java Technology-Based Memory Model," *Proceedings of the JavaOne Conference.* San Francisco, CA, June 2005.

[Bray 2008]
Bray, Tim, Paoli, Jean, Sperberg-McQueen, C.M., Maler, Eve, & Yergeau, Francois (eds). *Extensible Markup Language (XML) 1.0 (Fifth Edition).* http://www.w3.org/TR/xml/ (2008).

[Campione 1996]
Campione, Mary & Walrath, Kathy. *The Java Tutorial*. http://www.telecom.ntua.gr/HTML.Tutorials/index.html (1996).

[CCITT 1988]
CCITT. *CCITT Blue Book, Recommendation X.509 and IS0 9594-8: The Directory-Authentication Framework*. Technical Report, Geneva, 1988.

[Chan 1999]
Chan, Patrick, Lee, Rosanna, & Kramer, Douglas. *The Java Class Libraries: Supplement for the Java 2 Platform, v1.2, second edition, Volume 1*. Prentice Hall, 1999.

[Chess 2007]
Chess, Brian & West, Jacob. *Secure Programming with Static Analysis*. Addison-Wesley Professional, 2007.

[Christudas 2005]
Christudas, Binudlas. *Internals of Java Class Loading*. http://onjava.com/pub/a/onjava/2005/01/26/classloading.html (2005).

[Coomes 2007]
Coomes, Johm, Peter, Kessler, & Printezis, Tony. "Garbage Collection-Friendly Programming," *Proceedings of the JavaOne Conference.* San Francisco, CA, May 2007.

[Cunningham 1995]
Cunningham, Ward. "The CHECKS Pattern Language of Information Integrity," *Pattern Languages of Program Design*. Edited by James O. Coplien and Douglas C Schmidt. Addison-Wesley, 1995.

[Daconta 2000]
Daconta, Michael C. *When Runtime.exec() Won't.* http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-traps.html (2000).

[Daconta 2003]
Daconta, Michael C., Smith, Kevin T., Avondolio, Donald, & Richardson, W. Clay. *More Java Pitfalls*. Wiley Publishing Inc., 2003.

[Davis 2008]
Davis, Mark & Suignard, Michel. *Unicode Technical Report #36, Unicode Security Considerations.* http://www.unicode.org/reports/tr36/ (2008).

[Davis 2009]
Davis, Mark, Whistler, Ken, & Martin Dürst. *Unicode Standard Annex #15, Unicode Normalization Forms*. http://unicode.org/reports/tr15/ (2009).

[Dormann 2008]
Dormann, Will. *Signed Java Applet Security: Worse than ActiveX?*. http://www.cert.org/blogs/vuls/2008/06/signed_java_security_worse_tha.html (2008).

[Darwin 2004]
Darwin, Ian F. *Java Cookbook*. O'Reilly Media, 2004.

[Doshi 2003]
Doshi, Gunjan. *Best Practices for Exception Handling*. http://onjava.com/pub/a/onjava/2003/11/19/exceptions.html (2003).

[ESA 2005]
European Space Agency (EAS) Board for Software Standardisation and Control (BSSC). *Java Coding Standards.* ftp://ftp.estec.esa.nl/pub/wm/wme/bssc/Java-Coding-Standards-20050303-releaseA.pdf (2005).

[FindBugs 2008]
FindBugs. *FindBugs Bug Descriptions.* http://findbugs.sourceforge.net/bugDescriptions.html (2008).

[Fisher 2003]
Fisher, Maydene, Ellis, Jon, & Bruce, Jonathan. *DBC API Tutorial and Reference, 3rd edition.* Prentice Hall, The Java Series, 2003.

[Flanagan 2005]
Flanagan, David. *Java in a Nutshell, 5th edition.* O'Reilly Media, Inc., 2005.

[Fortify 2008]
Fortify Softwaree. *Fortify Taxonomy: Software Security Errors.* http://www.fortify.com/vulncat/en/vulncat/index.html (2008).

[Fox 2001]
Fox, Joshua. *When is a Singleton not a Singleton?* Sun Developer Network (SDN), 2001.

[Gafter 2006]
Gafter, Neal. *Thoughts about the Future of Java Programming.* http://gafter.blogspot.com/ (2006-2010).

[Gamma 1995]
Gamma, Erich, Helm, Richard, Johnson, Ralph, & Vlissides, John M. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1995.

[Garms 2001]
Garms, Jess & Somerfield, Daniel. *Professional Java Security*. Wrox Press Ltd., 2001.

[Goetz 2002]
Goetz, Brian. *Java theory and practice: Don't let the "this" reference escape during construction.* http://www.ibm.com/developerworks/java/library/j-jtp0618.html (2002).

[Goetz 2004a]
Goetz, Brian. *Java theory and practice: Garbage collection and performance*. http://www.ibm.com/developerworks/java/library/j-jtp01274.html (2004).

[Goetz 2004b]
Goetz, Brian. *Java theory and practice: The exceptions debate: To check, or not to check?* http://www.ibm.com/developerworks/java/library/j-jtp05254.html (2004).

[Goetz 2004c]
Goetz, Brian. *Java Theory and Practice: Going Atomic.* http://www.ibm.com/developerworks/java/library/j-jtp11234/ (2004).

[Goetz 2005a]
Goetz, Brian. *Java theory and practice: Be a good (event) listener, Guidelines for writing and supporting event listeners*. http://www.ibm.com/developerworks/java/library/j-jtp07265/index.html (2005).

[Goetz 2005b]
Goetz, Brian. *Java Theory and Practice: Plugging Memory Leaks with Weak References.* http://www.ibm.com/developerworks/java/library/j-jtp11225/ (2005).

[Goetz 2006]
Goetz, Brian, Pierels, Tim, Bloch, Joshua, Bowbeer, Joseph, Holmes, David, & Lea, Doug. *Java Concurrency in Practice*. Addison Wesley Professional, 2006.

[Goetz 2006b]
Goetz, Brian. *Java Theory and Practice: Good Housekeeping Practices.* http://www.ibm.com/developerworks/java/library/j-jtp03216.html (2006).

[Goetz 2006c]
Goetz, Brian. *Java theory and practice: Managing volatility, Guidelines for using volatile variables*. http://www.ibm.com/developerworks/java/library/j-jtp06197.html (2006).

[Goldberg 1991]
Goldberg, David. *What Every Computer Scientist Should Know About Floating-Point Arithmetic.* http://docs.sun.com/source/806-3568/ncg_goldberg.html (1991).

[Gong 2003]
Gong, LI, Ellison, Gary, & Dageford, Mary. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation, 2nd edition*. Prentice Hall, The Java Series, 2003.

[Gosling 2005]
Gosling, James, Joy, Bill, Steel, Guy, & Bracha, Gilad. *Java Language Specification, 3rd edition.* Addison Wesley, 2005.

[Grand 2002]
Grand, Mark. *Patterns in Java, Volume 1, Second Edition*. Wiley, 2002.

[Greanier 2000]
Greanier, Todd. *Discover the Secrets of the Java Serialization API.* http://java.sun.com/developer/technicalArticles/Programming/serialization/ (2000).

[Green 2008]
Green, Roedy. *Canadian Mind Products Java & Internet Glossary*. http://mindprod.com/jgloss/jgloss.html (2008).

[Grosso 2001]
Grosso, William. *Java RMI*. O'Reilly Media, 2001.

[Gupta 2005]
Gupta, Satish Chandra & Palanki, Rajeev. *Java Memory Leaks - Catch Me if You Can.* http://www.ibm.com/developerworks/rational/library/05/0816_GuptaPalanki/ (2005).

[Haack 2007]
Haack, Christian, Poll, Erik, Schafer, Jan, & Schubert, Alesky. *Immutable Objects for a Java-Like Language,"* 347-362. *Proceedings of the 16th European Conference on Programming.* Braga, Portugal. Springer-Verlag, 2007.

[Haggar 2000]
Haggar, Peter. *Practical Java Programming Language Guide.* Addison-Wesley Professional, 2000.

[Halloway 2000]
Halloway, Stuart. *Java Developer Connection Tech Tips.* http://javaservice.net/~java/bbs/read.cgi?m=devtip&b=jdc&c=r_p_p&n=954297433 **(2000).**

[Harold 1997]
Harold, Elliotte Rusty. *Java Secrets.* Wiley, 1997.

[Harold 1999]
Harold, Elliotte Rusty. *Java I/O.* O'Reilly Media, 1999.

[Harold 2006]
Harold, Elliotte Rusty. *Java I/O, 2nd Edition.* O'Reilly Media, 2006.

[Hawtin 2008]
Hawtin, Thomas. *Secure Coding Antipatterns: Preventing Attacks and Avoiding Vulnerabilities.* http://www.makeitfly.co.uk/Presentations/london-securecoding.pdf (2008).

[Henney 2003]
Henney, Kevlin. *Null Object, Something for Nothing*. http://www.two-sdg.demon.co.uk/curbralan/papers/europlop/NullObject.pdf (2003).

[Hitchens 2002]
Hitchens, Ron. *Java NIO*. O'Reilly Media, 2002.

[Hornig 2007]
Hornig, Charles. *Advanced Java Globalization.* http://developers.sun.com/learning/javaoneonline/2007/pdf/TS-2873.pdf (2007).

[Horstmann 2004]
Horstmann, Cay & Cornell, Gary. *Core Java 2 Volume I - Fundamentals, Seventh Edition*. Prentice Hall PTR, 2004.

[Hovemeyer 2007]
Hovemeyer, David & Pugh, William. "Finding more null pointer bugs, but not too many," *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. San Diego, CA. June 2007.

[Hunt 1998]
Hunt, J. & Long, F. *Java's reliability: an analysis of software defects in Java*. http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=00722326 (1998).

[IEC 2006]
International Electrotechnical Commission. *Analysis techniques for system reliability - Procedure for failure mode and effects analysis (FMEA)*, 2nd ed. (IEC 60812). IEC, January 2006.

[JSR-133 2004]
JSR-133. *JSR-133: Java Memory Model and Thread Specification.* http://www.cs.umd.edu/~pugh/java/memoryModel/jsr133.pdf (2004).

[Kabanov 2009]
Kabanov, Jevgengi. *The Ultimate Java Puzzler.* http://dow.ngra.de/2009/02/16/the-ultimate-java-puzzler/ (2009).

[Kabutz 2001]
Kabuts, Heinz M. *The Java Specialists' Newsletter*. http://www.javaspecialists.eu/archive/archive.jsp (2001).

[Kalinovsky 2004]
Kalinovsky, Ales. *Covert Java: Techniques for Decompiling, Patching, and Reverse Engineering*. SAMS Publishing, 2004.

[Knoernschild 2001]
Knoernschild, Kirk. *Java Design: Objects, UML, and Process*. Addison-Wesley Professional, 2001.

[Lai 2008]
Lai, C. "Java Insecurity: Accounting for Subtleties That Can Compromise Code,"*Software, IEEE 25*, 1 (Jan-Feb 2008): 13-19.

[Langer 2008]
Langer, Angelica. *Java Generics FAQ.* http://www.angelikalanger.com/GenericsFAQ/FAQSections/ProgrammingIdioms.html (2008).

[Lea 2000a]
Lea, Doug. *Concurrent Programming in Java, 2nd edition*. Addison Wesley, 2000.

[Lea 2000b]
Lea, Doug & Pugh, William. *Correct and Efficient Synchronization of Java Technology based Threads.* http://www.cs.umd.edu/~pugh/java/memoryModel/TS-754.pdf (2000).

[Lea 2008]
Lea, Doug. *The JSR-133 Cookbook for Compiler Writers.* http://g.oswego.edu/dl/jmm/cookbook.html (2008).

[Lee 2009]
Lee, Sangin, Somani, Mahesh, & Saha, Debashis. *Robust and Scalable Concurrent Programming: Lessons from the Trenches*. Oracle, 2009.

[Liang 1997]
Liang, Sheng. *The Java Native Interface, Programmer's Guide and Specification*. Addison-Wesley, 1997.

[Liang 1998]
Liang, Sheng & Bracha, Gilad. "Dynamic Class Loading in the Java Virtual Machine," *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Vancouver, BC, 1998, ACM, 1998.

[Lieberman 1986]
Lieberman, Henry. "Using prototypical objects to implement shared behavior in object-oriented systems," *Proceedings of the 1986 conference on Object-oriented programming systems, languages and applications,* Portland, ME, ACM, 1986.

[Lo 2005]
Lo, Chia-Tien Dan, Srisa-an, Witawas, & Chang, J. Morris. "Security Issues in Garbage Collection," *STSC Crosstalk* (October 2005).

[Long 2005]
Long, Fred. *Software Vulnerabilities in Java* (CMU/SEI -2004-TN-044). Software Engineering Institute, Carnegie Mellon University, 2005.
http://www.sei.cmu.edu/library/abstracts/reports/05tn044.cfm.

[Low 1997]
Low, Douglas. *Protecting Java Code via Obfuscation.*
http://www.cs.arizona.edu/~collberg/Research/Students/DouglasLow/obfuscation.html (1997).

[Macgregor 1998]
Macgregor, Robert, Durbin, Dave, Owlett, John, &Yeomans, Andrew. *Java Network Security*. Prentice Hall, 1998.

[Mak 2002]
Mak, Ronald. *Java Number Cruncher, The Java Programmer's Guide to Numerical Computing*. Prentice Hall, 2002.

[Manson 2004]
Manson, Jeremy & Goetz, Brian. *JSR 133 (Java Memory Model) FAQ*.
http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html (2004).

[Marco 1999]
Pistoia, Marco, Reller, Duane F., Gupta, Deepak, Nagnur, Milind, & Ramani, Ashok K. *Java 2 Network Security*. IBM Corporation (1999).

[Mcgraw 1998]
Mcgraw, Gary & Felten, Edward. *Twelve rules for developing more secure Java code.*
http://www.javaworld.com/javaworld/jw-12-1998/jw-12-securityrules.html (1998).

[Mcgraw 1999]
Mcgraw, Gary & Felten, Edward W. *Securing Java, Getting Down to Business with Mobile Code.* Wiley, 1999.

[Microsoft 2009]
Microsoft. *Using SQL Escape Sequences*. http://msdn.microsoft.com/en-us/library/ms378045%28SQL.90%29.aspx (2009).

[Miller 2009]
Miller, Alex. *Java Platform Concurrency Gotchas.* http://www.slideshare.net/alexmiller/java-concurrency-gotchas (2009).

[MITRE 2010]
MITRE. *Common Weakness Enumeration.* http://cwe.mitre.org/ (2010).

[Mocha 2007]
Mocha. *Mocha, the Java Decompiler.* http://www.brouhaha.com/~eric/software/mocha/ (2007).

[Muchow 2001]
Muchow, John W. *MIDlet Packaging with J2ME.* http://www.onjava.com/pub/a/onjava/2001/04/26/midlet.html (2001).

[Naftalin 2006]
Naftalin, Maurice & Wadler, Philip. *Java Generics and Collections.* O'Reilly Media, 2006.

[Netzer 1992]
Netzer, Robert H. & Miller, Barton, P. "What Are Race Conditions? Some Issues and Formalization." *ACM Letters on Programming Languages and Systems (LOPLAS) 1,1* (March 1992): 74-88.

[Neward 2004]
Neward, Ted. *Effective Enterprise Java.* Addison Wesley Professional, 2004.

[Nolan 2004]
Nolan, Godfrey. *Decompiling Java.* Apress, 2004.

[Oaks 1999]
Oaks, Scott & Wong, Henry. *Java Threads (2nd Edition.)* O'Reilly Media, 1999.

[Oaks 2001]
Oaks, Scott. *Java Security*. O'Reilly Media, 2001.

[Oaks 2004]
Oaks, Scott & Wong, Henry. *Java Threads (3rd Edition)*. O'Reilly Media, 2004.

[O'Reilly 2003]
O'Reilly Media. Java Enterprise Best Practices. 2003.

[OWASP 2007]
OWASP. *OWASP TOP 10 FOR JAVA EE.* https://www.owasp.org/images/8/89/OWASP_Top_10_2007_for_JEE.pdf (2007).

[OWASP 2008]
OWASP. *OWASP*. http://www.owasp.org/index.php/Main_Page (2008).

[Philion 2003]
Philion, Paul. *Beware the Dangers of Generic Exceptions.*
http://www.javaworld.com/javaworld/jw-10-2003/jw-1003-generics.html (2003).

[Pistoia 2004]
Pistoia, Marco, Nagaratnam, Nataraj, Koved, Larry, & Nadalin, Anthony. *Enterprise Java Security: Building Secure J2EE Applications.* Addison Wesley, 2004.

[Pugh 2004]
Pugh, William. *The Java Memory Model (discussions reference).*
http://www.cs.umd.edu/~pugh/java/memoryModel/ (2004).

[Pugh 2008]
Pugh, William. *Defective Java Code: Turning WTF Code into a Learning Experience.*
http://72.5.124.65/learning/javaoneonline/j1sessn.jsp?sessn=TS-6589&yr=2008&track=javase (2008).

[Reasoning 2003]
Reasoning Inspection Service. *Defect Data Tomcat v 1.4.24*.
http://www.reasoning.com/pdf/Tomcat_Defect_Report.pdf (2003).

[Rotem-Gal-Oz 2008]
Rotem-Gal-Oz, Arnon. *Fallacies of Distributed Computing Explained.*
http://www.rgoarchitects.com/Files/fallacies.pdf (2008).

[Roubtsov 2003a]
Roubtsov, Vladimir. *Breaking Java Exception-Handling Rules is Easy.*
http://www.javaworld.com/javaworld/javaqa/2003-02/02-qa-0228-evilthrow.html (2003).

[Roubtsov 2003b]
Roubtsov, Vladimir. *Into the Mist of Serialization Myths.*
http://www.javaworld.com/javaworld/javaqa/2003-06/02-qa-0627-mythser.html?page=1 (2003).

[Schneier 2000]
Schneier, Bruce. *Secrets and Lies—Digital Security in a Networked World*. John Wiley and Sons, 2000.

[Schildt 2007]
Schildt, Herb. *Herb Schildt's Java Programming Cookbook*. McGraw-Hill, 2007.

[Schoenefeld 2004]
Schoenefeld. (Nov. 2004). *Java Vulnerabilities in Opera 7.54 BUGTRAQ Mailing List* [online].
Available email: bugtraq@securityfocus.com.

[Schwarz 2004]
Schwarz, Don. *Avoiding Checked Exceptions*. http://www.oreillynet.com/onjava/blog/2004/09/avoiding_checked_exceptions.html (2004).

[Schweisguth 2003]
Schweisguth, Dave. *Java Tip 134: When catching exceptions, don't cast your net too wide.* http://www.javaworld.com/javaworld/javatips/jw-javatip134.html?page=2 (2003).

[Seacord 2005]
Seacord, Robert C. *Secure Coding in C and C++*. Addison-Wesley, 2005.

[Sen 2007]
Sen, Robi. *Avoid the dangers of XPath injection.* http://www.ibm.com/developerworks/xml/library/x-xpathinjection.html (2007).

[Sun 1999a]
Sun Microsystems, Inc. *Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated?* http://java.sun.com/j2se/1.4.2/docs/guide/misc/threadPrimitiveDeprecation.html (1999).

[Sun 1999b]
Sun Microsystems, Inc. *The Java Virtual Machine Specification.* http://java.sun.com/docs/books/jvms/ (1999).

[Sun 1999c]
Sun Microsystems, Inc. *Code Conventions for the Java Programming Language.* http://java.sun.com/docs/codeconv/ (1999).

[Sun 2000]
Sun Microsystems, Inc. *Java 2 SDK, Standard Edition Documentation.* http://java.sun.com/j2se/1.3/docs/guide/ (1995-2000).

[Sun 2002]
Sun Microsystems, Inc. *Default Policy Implementation and Policy File Syntax, Document revision 1.6*. http://java.sun.com/javase/6/docs/technotes/guides/security/PolicyFiles.html (2002).

[Sun 2003a]
Sun Microsystems, Inc. *Jar File Specification.* http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html (2003).

[Sun 2003b]
Sun Microsystems, Inc. *Sun ONE Application Server 7 Performance Tuning Guide.* http://docs.sun.com/source/817-2180-10/ (2003).

[Sun 2004]
Sun Microsystems, Inc. *Java Platform Debugger Architecture (JPDA).* http://java.sun.com/javase/6/docs/technotes/guides/jpda/index.html (2004).

[Sun 2004b]
Sun Microsystems, Inc. *Generics.*
http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html (2004).

[Sun 2006a]
Sun Microsystems, Inc. *Java Platform, Standard Edition 6 Documentation.*
http://java.sun.com/javase/6/docs/index.html (2006).

[Sun 2006b]
Sun Microsystems, Inc. *Java Virtual Machine Tool Interface (JVM TI).*
http://java.sun.com/javase/6/docs/technotes/guides/jvmti/index.html (2006).

[Sun 2006c]
Sun Microsystems, Inc. *Java 2 Platform Security Architecture.*
http://java.sun.com/javase/6/docs/technotes/guides/security/spec/security-spec.doc.html (2006).

[Sun 2006d]
Sun Microsystems, Inc. *Java Security Guides.*
http://java.sun.com/javase/6/docs/technotes/guides/security/ (2006).

[Sun 2006e]
Sun Microsystems. *Supported Encodings.*
http://java.sun.com/javase/6/docs/technotes/guides/intl/encoding.doc.html (2006).

[Sun 2006f]
Sun Microsystems, Inc. *Monitoring and Management for the Java Platform.*
http://java.sun.com/javase/6/docs/technotes/guides/management/index.html (2006).

[Sun 2006g]
Sun Microsystems, Inc. *Java Platform, Standard Edition 6.*
http://java.sun.com/javase/6/docs/technotes/guides/management/toc.html (2006).

[Sun 2008a]
Sun Microsystems, Inc. *The Java Tutorials.* http://java.sun.com/docs/books/tutorial/index.html (2008).

[Sun 2008b]
Sun Microsystems, Inc. *SUN Developer Network.* http://developers.sun.com/ (1994-2008).

[Sun 2008c]
Sun Microsystems, Inc. *JDK 7 Documentation.* http://download.java.net/jdk7/docs/ (2008).

[Sun 2008d]
Sun Microsystems, Inc. *Java Security Architecture.*
http://java.sun.com/j2se/1.5.0/docs/guide/security/spec/security-specTOC.fm.html (2008).

[Sun 2008e]
Sun Microsystems, Inc. *Java Plug-in and Applet Architecture.* http://java.sun.com/javase/6/docs/technotes/guides/jweb/applet/applet_execution.html (2008).

[Sun 2009a]
Sun Microsystems, Inc. *Secure Coding Guidelines for the Java Programming Language, Version 3.0.* http://java.sun.com/security/seccodeguide.html (2009).

[Sun 2009b]
Sun Microsystems. *Java Platform, Standard Edition 6 API Specification*. http://java.sun.com/javase/6/docs/api/ Sun Microsystems, Inc. (2009).

[Sun 2010a]
Sun Microsystems, Inc. *Java SE 6 HotSpot Virtual Machine Garbage Collection Tuning,* http://java.sun.com/javase/technologies/hotspot/gc/gc_tuning_6.html (2010).

[Sun 2010b]
Sun Microsystems, Inc. *java - the Java application launcher.* http://java.sun.com/javase/6/docs/technotes/tools/windows/java.html (2006).

[Steel 2005]
Steel, Christopher, Nagappan, Ramesh, & Lai, Ray. *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*. Prentice Hall, 2005.

[Steuck 2002]
Steuck, Gregory. *XXE (Xml eXternal Entity) attack.* http://www.securityfocus.com/archive/1/297714 (2002).

[Sutherland 2010]
Sutherland, Dean F. & Scherlis, William L. "Composable thread coloring," *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*. Bangalore, India, 2010, ACM, 2010.

[Tanenbaum 2002]
Tanenbaum, Andrew S. & Van Steen, Maarten. *Distributed Systems: Principles and Paradigms, 2/E*. Prentice Hall, 2002.

[Venners 1997]
Venners, Bill. *Security and the Class Loader Architecture.* http://www.javaworld.com/javaworld/jw-09-1997/jw-09-hood.html?page=1 (1997).

[Venners 2003]
Venners, Bill. *Failure and Exceptions, A Conversation with James Gosling, Part II*. http://www.artima.com/intv/solid.html (2003).

[Wheeler 2003]
Wheeler, David A. *Secure Programming for Linux and Unix HOWTO*. http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/index.html (2003).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	May 2010	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Java Concurrency Guidelines	FA8721-05-C-0003

6. AUTHOR(S)

Fred Long , Dhruv Mohindra , Robert Seacord , David Svoboda

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2010-TR-015

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	ESC-TR-2010-015

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. abstract (maximum 200 words)

An essential element of secure coding in the Java programming language is well-documented and enforceable coding standards. Coding standards encourage programmers to follow a uniform set of guidelines determined by the requirements of the project and organization, rather than by the programmer's familiarity or preference. Once established, these standards can be used as a metric to evaluate source code (using manual or automated processes).

The CERT Oracle Secure Coding Standard for Java provides guidelines for secure coding in the Java programming language. The goal of these guidelines is to eliminate insecure coding practices and undefined behaviors that can lead to exploitable vulnerabilities. Applying this standard will lead to higher quality systems that are robust and more resistant to attack.

This report documents the portion of those Java guidelines that are related to concurrency.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Java, concurrency, software security, coding standard, coding guidelines	213

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL